Understanding Software Maintenance in the Context of Software Architecture Evolution

A thesis submitted in partial fulfilment

of the requirements for the Degree

of

Master of Engineering in Software Engineering

by

Chathrie Wimalasooriya

University of Canterbury

2019

Abstract

Context and background: Software maintenance and evolution occur throughout the lifetime of a software system and involve activities such as fixing bugs, adding functionality or improving the software's design. Maintenance and evolution consume the majority of a software product's lifecycle costs. With system evolution, the software architecture of a system evolves as well.

Problem: A poor understanding of software maintenance tasks can negatively impact the software architecture of a system. For example, practitioners may lack sufficient details to interpret the effort and implications of performing a task. From an architecture perspective, practitioners need to understand where and how much architecture changes happen. Poorly performed or understood maintenance tasks may lead to the accumulation of technical debt and compromise quality attributes of systems (e.g., understandability, extensibility, reliability, etc.). Therefore, this thesis aims at better understanding software maintenance in the context of architecture evolution. In detail, the research questions investigated in this thesis are: RQ1 – What is the state-of-the-art of maintenance-related tasks? RQ2 – How do software architectures change during maintenance and evolution?

Research method: To answer RQ1 and to analyse and characterize maintenance tasks, we conducted a systematic mapping study analysing the literature from January 2010 to November 2017. To answer RQ2 and to explore architecture evolution at system and component level, we conducted a replication of an empirical study that analyses the evolution path of several open source software systems. We analysed the evolution patterns between different types of version pairs (e.g., between major and minor releases). Our replication included more (and more recent) versions of software system included in the original study.

Findings and conclusions: The answer to RQ1 offers a) a hierarchical classification of software maintenance tasks, and b) a catalogue of characteristics of tasks (what do tasks impact [e.g., source code, documentation], when would they be performed, etc.). When characterising maintenance tasks based on the software development artefacts they impact, we found that architectural impact appears most frequently. Also, characterizing maintenance tasks based on their impact on quality attributes is another frequent way of characterising tasks. The answer to RQ2 not only confirmed findings from the original study and therefore strengthens the empirical evidence related to architectural evolution, but also identified reasons for why differences in evolution patterns exist. Furthermore, our replication showed that architectures do not stabilize over time, even if longer evolution paths are considered.

Acknowledgements

Finishing a Master's thesis is a collaborative work of people who have accompanied me in overcoming numerous obstacles during my research. I am deeply grateful to all of them and take this moment to express my gratitude.

First and foremost, I want to thank my supervisor, Matthias Galster for the opportunity to carry out this research at the University of Canterbury. I am extremely grateful to him for his patience, support, guidance, and encouragement during this research. More importantly, I was a novice in conducting research and academic writing when joining as a Master's student. With his immense dedication and continuous feedback, I was able to enhance my knowledge, complete the research work and thesis. I appreciate his precious advices, especially on making my arguments concrete and following a logical flow which is beneficial not only to complete a Master's degree but also for my academic career in my entire life.

Extending my special thanks go to Nenad Medvidovic, Duc Minh Le and Daniel Link at the University of Southern California, USA for sharing their knowledge with us about their empirical studies on architecture evolution and also for providing required information including their tools, source code and experimental data to conduct this research.

I want to extend the gratitude towards the members of the Department of Computer Science and Software Engineering, College of Engineering and Student Care at the University of Canterbury who played vital roles in achieving this milestone. Besides, I would like to acknowledge my gratitude to the Sabaragamuwa University of Sri Lanka for providing the funding to finish my degree.

This accomplishment would not have been possible without all of your assistance and guidance. I am grateful to all of you. Thank you.

Chathrie Wimalasooriya

Table of Contents

List of Figures ..... **vi**

List of Tables ..... **vii**

1 Introduction **1**

1.1 Problem statement and motivation 1
1.2 Research questions 5
1.3 Research approach 9
1.4 Thesis contributions 11
1.5 Overview of the thesis 11

2 Background **13**

2.1 Software maintenance 13
2.2 Software architecture 14
2.3 Software architecture decay 17

3 A Systematic Mapping Study on Software Maintenance Tasks and their Characteristics **19**

3.1 Introduction 19
3.2 Existing secondary studies 20
3.3 Related work 21
3.3.1 Classification of maintenance 21
3.3.2 Classification of change 23
3.4 Mapping study research questions 25
3.5 Study design 26
3.5.1 Search scope 26
3.5.2 Search strategy and search string 27
3.5.3 Study selection 29
3.5.4 Data extraction and analysis 31
3.6 Results 31
3.6.1 Overview of selected papers 31
3.6.2 Software maintenance tasks (RQ 1.1) 32

3.6.2.1 Overview of taxonomy ... 33
3.6.2.2 Frequently reported types of maintenance tasks ... 37
3.6.2.3 Frequently reported maintenance tasks ... 39
3.6.3 Characteristics of maintenance tasks (RQ 1.2) ... 40
3.7 Discussion ... 42
3.7.1 Frequently reported maintenance tasks ... 42
3.7.2 Characteristics of maintenance tasks ... 42
3.7.3 Comparison to existing classification of software maintenance ... 47
3.7.4 Implications for researchers and practitioners ... 49
3.8 Threats to validity ... 50
3.9 Conclusions ... 52
4 Architecture Decay in Open Source Software Systems: an External Replication .. 54
4.1 Introduction ... 54
4.2 Replications in software engineering ... 56
4.3 Background and related work ... 57
4.3.1 Architecture recovery ... 57
4.3.2 Architecture change metrics ... 60
4.3.3 Replications of studies on architecture evolution ... 60
4.4 Original Study ... 61
4.4.1 Research questions of original study ... 61
4.4.2 Architecture recovery techniques ... 62
4.4.3 Architecture change metrics ... 64
4.4.4 Original study design and execution ... 66
4.5 Types of version pairs ... 68
4.6 Major findings of original study ... 69
4.7 External replication ... 70
4.7.1 Interaction with original researchers ... 70
4.7.2 Changes to the original study ... 70
4.7.3 Research questions of replication ... 72
4.7.4 Replication procedure ... 72
4.8 Comparison of results ... 73
4.8.1 RQ2.1: Architectural changes at system-level ... 74
4.8.1.1 Full comparison ... 75

4.8.1.2 Exact comparison 77
4.8.1.3 Summary of RQ2.1 86
4.8.2 RQ2.2: Architectural changes at component-level 86
4.8.2.1 Full comparison 87
4.8.2.2 Exact comparison 88
4.8.2.3 Summary of RQ2.2 93
4.9 Discussion of results 94
4.9.1 Comparison with original study 94
4.9.2 Lessons learned about replications 96
4.10 Threats to validity 97
4.11 Conclusions and future works 98
4.11.1 Conclusions 98
4.11.2 Future work 100
5 Conclusions 101
5.1 Answers to research questions 101
5.1.1 RQ1: What is the state-of-the-art of maintenance-related tasks? 101
5.1.2 RQ2: How do software architectures change during maintenance and evolution? 102
5.2 Thesis contributions and discussion 103
5.3 Future work 106
References 108
Appendix 117
Appendix A 117
Appendix B 118
Appendix C 124
Appendix D 125
Appendix E 127

List of Figures

Figure 1: Relationship between research questions 6
Figure 2: Architectural change at system-level during system evolution 8
Figure 3: Architectural change at component-level during system evolution 9
Figure 4: Decision tree proposed by Chapin for classifying maintenance types 22
Figure 5: Study selection process 30
Figure 6: Distribution of papers over publication type and year 32
Figure 7: Distribution of papers over time period 32
Figure 8: Overview of taxonomy for software maintenance tasks: main types 33
Figure 9: Maintenance type “Bug fixing” (MT1) 34
Figure 10: Maintenance type “Feature enhancement/modification” (MT2) 34
Figure 11: Maintenance type "Feature removal" (MT3) 34
Figure 12: Maintenance type "Feature addition" (MT4) 35
Figure 13: Maintenance type “Quality improvement” (MT5) 35
Figure 14: Maintenance type “Pre-change activities” (MT6) 35
Figure 15: Maintenance type “Post-change activities” (MT7) 35
Figure 16: Maintenance type “Documentation modification” (MT8) 36
Figure 17: Maintenance type “Refactoring” (MT9) 36
Figure 18: Distribution of studies over maintenance type and publication year 38
Figure 19: ARACDE's workflow related to this study [27] 68
Figure 20: Replication procedure 73
Figure 21: Analysis conducted to answer RQ2.1 74
Figure 22: *a2a* values for PDFBox 83
Figure 23:*a2a* values for Xerces 84
Figure 24: *cvg* (*s*, *t*) values for Xerces 90
Figure 25: *cvg* (*t*, *s*) values for Xerces 91

List of Tables

Table 1: Search strings 29
Table 2: Data items extracted from selected studies 31
Table 3: Most frequently reported maintenance tasks 39
Table 4: Characteristics of maintenance tasks 41
Table 5: Cross-tabulation of maintenance tasks and characteristic 1 43
Table 6: Cross-tabulation of maintenance tasks and characteristic 2 44
Table 7: Cross-tabulation of maintenance tasks and characteristic 3 44
Table 8: Cross-tabulation of maintenance tasks and characteristic 4 45
Table 9: Tool support for software maintenance tasks 47
Table 10: Analysed systems 71
Table 11: Average *a2a* values (as percentage) 75
Table 12: Average *a2a* values for exact comparison 78
Table 13: Comparison of cluster files based on text similarity 79
Table 14: Architecture similarity of cluster files 81
Table 15: Similarity of trends (*a2a*) in original study and replication 83
Table 16: Statistical tests of differences in *a2a* values in original study and replication 85
Table 17: Average *cvg* values 88
Table 18: Average *cvg* values for exact comparison 89
Table 19: Similarity of trends (*cvg*) in original study and replication 89
Table 20: Statistical test results for *cvg* (*s*, *t*) 92
Table 21: Statistical test results for *cvg* (*t*, *s*) 93

1 Introduction

1.1 Problem statement and motivation

Every software system has an architecture. The software architecture can be considered as the blue print of any software and describes the high-level structure and major design decisions (see Section 2.2 for a more detailed discussion on the concept of software architecture). The quality of the architecture impacts the development and evolution of the software [1]. On the other hand, the architecture is impacted by software evolution. Evolution of a system occurs throughout its lifetime [2] and involves all activities required to maintain, improve and extend a software product, such as bug fixing, adding functionality, revising design, improving design, etc. Research has found that the cost of software evolution and maintenance consume the majority of the overall product lifecycle costs, i.e., around 40%-80% [3-6] and are difficult and time-consuming [2, 7, 8].

With system growth and evolution, the architecture of the software evolves as well and the complexity of the software architecture increases. Increasing architecture complexity has to be taken into account, because architecture is not static or isolated, but defines how components are connected, interactions between components, interactions between components and system environment and also the principles guiding its design and evolution. Component can include high-level components (e.g., modules, packages), but also low-level components and implementation-level entities (e.g., source files, classes, or methods) [9]. Since an architecture consists of multiple components, changes to an architecture are not necessarily local (i.e., impact not only one component), but could impact the behaviour of other components and the overall system behaviour [2].

With this increase of complexity, systems become less understandable and therefore more difficult to maintain (which eventually leads to architectural decay). Careless changes to a system, its architecture and code may violate initial and major architecture design decisions. For example, if a design decision is made to use a three-tier architecture, developers may not follow that pattern but take short-cuts and bypass the middle layer. These careless changes violate the original architecture and make the architecture difficult to comprehend due to the mismatch between the actual design and its original, intended and well-thought through. As a consequence, the behavior and quality of software may decrease (e.g., performance or security may suffer) and the software becomes more difficult to maintain, i.e., the probability

of introducing new bugs and the cost for maintenance increases [2, 10-12]. Therefore, to improve the quality of the (architectural) design, systems may have to undergo a major reengineering or reverse engineering or face early retirement. Also, the evolved architecture must ensure that the system is still flexible and continues to function as expected. Below we present concrete problems of maintenance.

- **P1: Difficulty of handling maintenance tasks in general**
  - **P1.1: Incorrect/inconsistent terminology**
    Different stakeholders use different terminology to refer same type of tasks or same terminology for different types of tasks. As argued by Herzig et al. [13], developers have different views on separating maintenance tasks. For example, practitioners may refer to a task as bug fixing, but it may actually involve other maintenance tasks such as enhancements of functionality or refactoring. Using such incorrect terminology or naming tasks inappropriately can lead to misinterpretation. This can result in misunderstanding maintenance tasks (e.g., a task being considered trivial) and could then affect managerial decisions, such as decisions related to estimating the required maintenance effort.
  - **P1.2: Lack of sufficient details about maintenance tasks**
    The level of detail used to describe a task is also important for management decisions such as allocating resources (experts, time, etc.), outsourcing maintenance activities, etc. More generic descriptions of a task may cause misunderstanding of what the maintenance task actually means. For example, if a maintenance task is described as “fix a bug”, the level of detail is too generic, as this task can have multiple implications, such as changing the visibility of a method, changing an algorithm inside a method, etc. In this situation, the work required to fix a bug may be underestimated in situations when it requires changing an algorithm compared to changing a data type within a method.

- **P2: Difficulty of handling architecture-related maintenance tasks**
  - **P2.1: Poor understanding of where architecture changes happen**
    A system’s architecture evolves constantly during its lifetime. Since evolution of systems causes increasing architecture complexity and affects quality attributes such as understandability, reliability, etc., software engineers must carefully handle the maintenance tasks related to architectural changes. In order to handle changes carefully

as well as to estimate the effort of architectural changes, software engineers are required to identify and predict where major architecture changes occur (at different levels of abstraction such as component and/or system level). However, they often lack knowledge regarding potential architectural changes since they look architectural changes only at the overall system level. This is not enough to understand architecture changes architecture-relevant changes may also occur at component level.

- **P2.2: Poor understanding of how much architecture change happens**
  Knowing only where the changes happen is not enough to understand architectural changes effectively. Software engineers need insights about how much (i.e., to which extent and when) architecture changes happen over time at both component level and system level. This information is necessary to understand how a change potentially impacts the architecture, trends of change over time and the stability of the architecture over time.

Previous studies have explored the evolution of software systems and only few of them analyse architecture evolution. Also, we still lack empirical data related to architecture evolution in particular *where* (P2.1) and *how* (P2.2) architecture changes occur. This would provide insights that help practitioners make more confident and informed decisions about maintenance and evolution.

The following are some practical consequences of above problems:

- **Difficulty of budgeting, staffing, resource allocation:** Managers involved in software maintenance often misunderstand what maintenance is and therefore deal with difficulties in budgeting, staffing and allocating resources to different kinds of development activities. This misunderstanding could be due to incorrect terminology (the problem explained as P1.1) used by different stakeholders (developers, managers), the generic level of detail (the problem explained as P1.2) that is used to describe a maintenance task, and also a poor understanding of design-related changes, such as how much architectural changes occur (the problem explained as P2.2).

- **Difficulty of handling cross-cutting tasks:** The consequences become worse when maintenance tasks are cross-cutting. With modern iterative development practices, maintenance tasks have to be organized and integrated in iterations. There can be maintenance tasks performed at both iteration level and across iterations (e.g., releases) [14]. Tasks related to maintenance may deal with the technical or architectural design of a system and not directly allocated to one iteration or to one sprint because those design issues may be cross-cutting several sprints. Some maintenance tasks such as correcting defects or adding new features can be allocated to one development iteration, but some tasks related to restructuring cannot be allocated to an iteration or release. For example, restructuring may involve engineers to re-think the design of the system and it is difficult to split this task into small steps as suggested by agile software development practices [15]. One reason for this difficulty could be using high abstract level tasks description as discussed in P1.2. Having concrete lower-level tasks to capture maintenance activities make easier to split tasks across iterations.
- **Difficulty of prioritizing tasks:** In order to handle urgent and continuous customer requests, managers have to prioritize tasks. Using too generic tasks descriptions as explained in P1.2 also causes difficulties in prioritizing tasks. The more concrete the task, the easier it is to understand the impact and complexity of change. This information can then be used to prioritize the tasks. For example, a bug caused by incorrect visibility level of the method can be fixed first before fixing the bug caused by the incorrect algorithm (considering the complexity of the task). Furthermore, when prioritizing tasks, development teams may focus on producing value (requested changes/functionalities) for their customers. Tasks related to technical or architectural design may receive lower priority and easily ignore negative implications of design decisions on the existing system. Also, maintenance teams or project managers who are responsible for managing and prioritizing tasks may lack knowledge related to identifying potential design related changes and technical challenges that the maintenance team must overcome and then easily ignore good practices of architecture/design evolution. This can lead to technical debt overtime [16]. Technical debt is a metaphor to indicate “not quite right code which we postpone making it right” [17].

- **Accumulation of technical debt:** Technical debt or poor design and code such as code smells, error-prone code (e.g., code that may introduce uncaught exceptions [18]), etc. [17, 19] has to be paid back at some point later in the lifetime of a system to keep the system maintainable and to meet business goals. Technical debt is paid especially with refactoring and rearchitecting [20]. However, the technical or architectural design of an evolving system cannot be ignored because maintaining the system is not only about producing values to the customers, but also about improving design to ensure long-term maintainability. Lehman argues that rearchitecting is inevitable and software systems have to change if systems should remain useful [21]. These changes should be carefully handled, which requires a good understanding of when (see P2.1) and how (see P2.2) architecture changes occur.
- **Architecture decay:** Architecture decay negatively impacts quality attributes such as understandability, testability, extensibility, reusability[22] reliability [23], etc. and eventually result in early retirement of software systems. Not being able to properly manage architecture decay cause architecture technical debt which is part of the overall technical debt [24, 25] complicates software maintenance [25]. Xiao et al. [26] show that a significant portion of the overall maintenance effort is consumed by paying interest on architecture debt. To avoid unexpected erosion and to anticipate cost and effort of architectural evolution, it is necessary to identify and track decay across the life time of a system which requires a good understanding of the impact and extent (i.e., the amount of architecture changes as explained in P2.2) of architectural change and nature of architecture decay [2, 27].

1.2 Research questions

Based on problems P1 and P2 discussed above, this section explains the research questions (RQ) addressed in this thesis, and how they are related to each other. The research questions are shown in Figure 1.

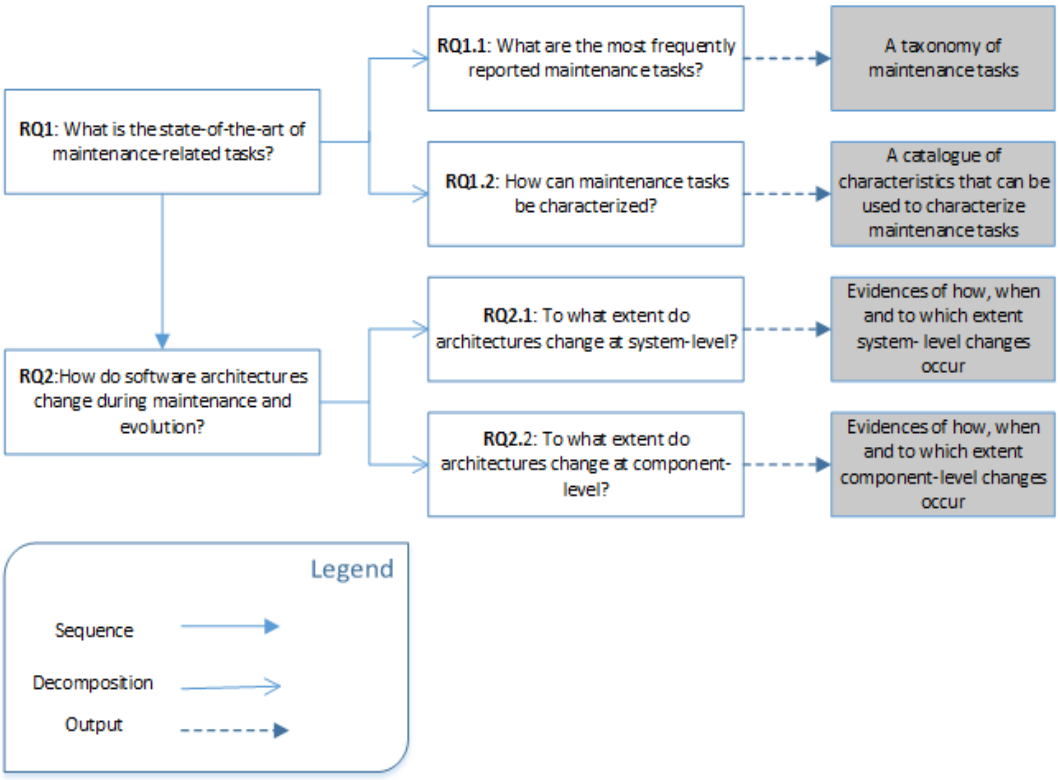


Figure 1: Relationship between research questions

To address the problems mentioned in the previous section, developers need a way to understand the effects of a change prior to making it. **Therefore, this thesis aims at better understanding software maintenance in the context of architecture evolution.** However, before we look deep into architecture evolution, we first need to investigate software maintenance in more detail. Therefore, we first, we need to identify and understand what software maintenance tasks are. This leads to the first research question addressed in this thesis:

RQ1: What is the state-of-the-art of maintenance-related tasks? RQ1 is concerned with obtaining a comprehensive understanding of software maintenance tasks. This question is decomposed into two sub-questions.

RQ1.1: What are the most frequently reported maintenance tasks? RQ1.1 extracts concrete maintenance tasks from the current literature on software maintenance and classifies these tasks into types of maintenance tasks (e.g., bug fixing, refactoring etc.). The classification can provide properly named tasks that can be used as a common taxonomy. This helps overcome difficulties such as using inconsistent terminology as discussed as P1.1 above. Also, this taxonomy describes

maintenance tasks in a hierarchical structure, i.e., it shows how high-level tasks can be split into concrete maintenance tasks to address P1.2.

RQ1.2: How can maintenance tasks be characterized? Based on analysing the current literature on maintenance tasks and based on further analysing maintenance tasks described in the literature, RQ1.2 provides a catalogue of characteristics that can characterize a maintenance task (e.g., based on artefacts impacted by a task, component-level change, system-level change, etc.). This catalogue is also in a hierarchical structure displaying grouping of characteristics that form more high-level characteristics. This could be used as a framework to evaluate and characterize maintenance tasks extracted in RQ1.1. Characterized tasks provide a more effective support in software maintenance. More importantly, characteristics of task help identify maintenance tasks related to architectural changes (how maintenance tasks will impact the architecture, how the change aligns with the existing architecture, etc.). For example, tasks characterized as having component-level impact potentially change the architecture. Such characteristics provide insights about change impact, difficulty and required effort for a particular task. Also, based on such characteristics, practitioners can pay more attention to such tasks and perform those tasks with more care to avoid architecture decay.

After gaining good understanding of software maintenance tasks, we need to understand the actual impact of change on the architecture. Therefore, we formulate a second research question:

RQ2: How do software architectures change during maintenance and evolution? RQ2 analyses architectural changes to help us understand software maintenance in the context of architecture evolution. RQ2 is decomposed into two sub-questions.

RQ2.1: To what extent do architectures change at system-level? RQ2.1 investigates architectural changes at system-level. System-level changes involve adding or removing implementation-level entities inside architecture components, or moving implementation-level entities between components, or adding and removing components themselves. Here, a component (e.g., a package in Java) is a group of implementation-level entities (e.g., a class or method in Java). For example, assume that in Figure 2, A1 and A2 are architectures of different versions of the same system, C1, C2, C3, C4 and C5 are components, and e1, e2, e3, …, e13 are implementation level entities inside these components.

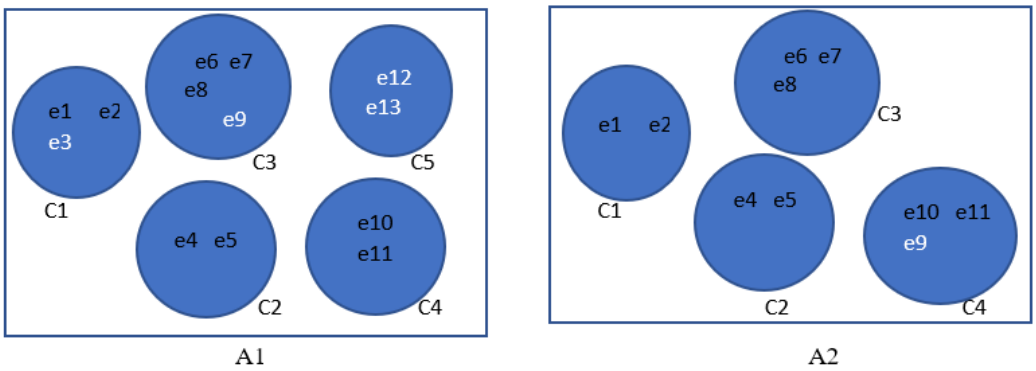


Figure 2: **Architectural change at system-level during system evolution**

The extent of change between A1 and A2 at system-level can be measured based on the number of changes to the architecture needed to transform A1 to A2. For example, during the evolution from A1 to A2, the following changes occur: e3 is removed, e9 is moved from C3 to C4, and component C5 is removed completely. To measure the changes at system-level, changes inside a component (i.e., removal of e3) as well as the changes between components (moving e9 and removing component C5) are considered. RQ2.1 focuses on such changes between a system's architectures during its development and evolution with respect to when, where and to what extent these changes happen.

RQ2.2: To what extent do architectures change at component-level? RQ2.2 investigates architectural changes at component-level. Figure 3 shows an example of the evolution of an architecture from A1 to A2 and changes at component-level. B*i* represents components of A1 and C*j* represents components of A2. As before, e1, e2, etc. represent implementation-level entities.

At component-level, architectural change is about the change in components that are common in both architectures before and after evolving from A1 to A2 (i.e., the components whose implementation-level entities match the implementation-level entities of a component in A2) or components added after evolving from A1 to A2. The similarity between the components before and after evolving, i.e., the similarity between B*i* and C*j* (see percentages on lines between components) can be measured based on the number of entities that are common in both components. If B*i* and C*j* are similar, this means that the same component that existed in A1 still exists in A2. For example, the similarity between B1 and C1 is 25% since e1 is common (i.e., one out of four entities are common to both B1 and C1). B1 has no common entities with C4, hence the similarity between B1 and C4 is 0%.

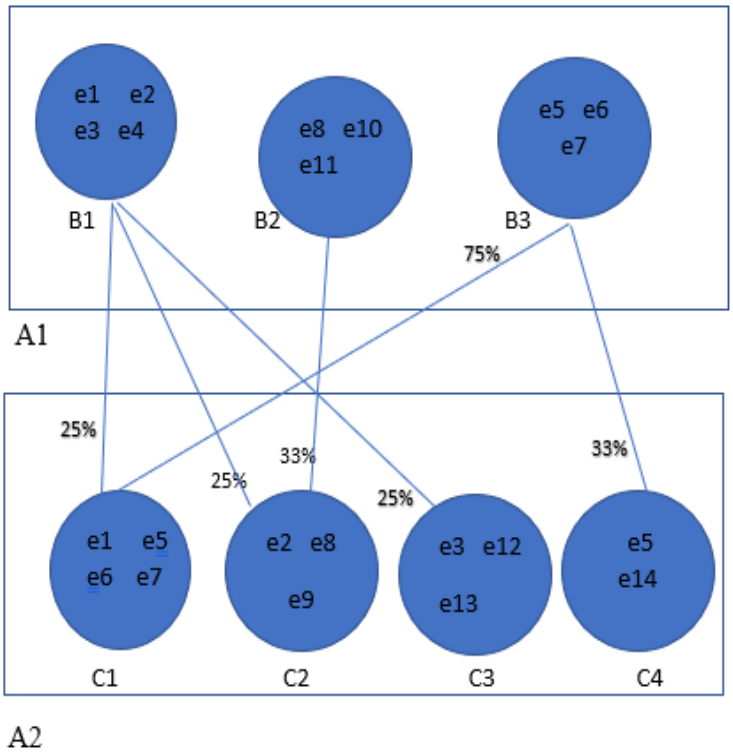


Figure 3: Architectural change at component-level during system evolution

In brief, RQ2.1 and RQ2.2 support gaining a better understanding of where architecture changes happen (whether in overall system architecture or in components) addressing the problem explained as P2.1, when these changes happen (by analysing architectural changes between different types of versions such as major releases and minor releases of a system) and to what extent architectural changes occur (by analysing a full release path of a system), addressing the problem explained as P2.2.

1.3 Research approach

To answer the two research questions, this thesis used following empirical research methods.

- **Systematic mapping study**: To answer RQ1, we conducted a systematic mapping study (sometimes referred to as scoping studies), i.e., a form of literature review [28] . A mapping study aims at structuring the research area under investigation through classifying existing studies and quantifying studies in each category of the classification. It focuses on identifying, evaluating and interpreting the research on a particular topic and on

extracting detailed topics covered in literature. In detail, the mapping study synthesizes software maintenance tasks discussed in the research literature and their characteristics and classifies literature based on these tasks and characteristics. A mapping study includes a systematic search process and study selection as well as data analysis. Mapping studies typically consist of several activities related to planning, conducting, and reporting of mapping studies. Petersen et al. [28] proposed updated guidelines for conducting a systematic mapping study while combining existing guidelines mostly from Kitchenham and Charters [29] and Petersen et al. [30]. They also include the guideline for evaluating mapping studies. In contrast to a systematic literature review [29], a mapping study can be done when there are fewer existing studies and when the research topic is broad. Also, different methods for data extraction and analysis would apply to systematic literature reviews. The details about our mapping study and its study design will be discussed in Chapter 3.

- **Empirical analysis of architecture evolution:** To answer RQ2, we conducted an empirical study of architectural changes in open source software systems. In particular, we conducted a replication of a large empirical study conducted previously by others to measure architectural changes across different versions of a software system. Generally, replication refers to a repetition of a study to verify or enhance the results of the original study [31]. Gomez et al. [32] define repetition as a type of replication, i.e., a new run of the original study with the same study settings (same researcher, protocol, site etc.) as in the original study, but with different data samples of the same populations (for example, picking a different set of practitioners from the population of software developers). In contrast to repetitions, settings of replications can be changed. For example, different researchers can run the experiment with the same protocol on a different population [32].

  The concept of replication plays a key role that allows enriching the body of software engineering knowledge. The results or outcome of empirical studies may not be reliable since, since it is not clear whether results of a study were produced by chance, accidentally or due to experimental configuration [32]. Replications can be grouped into two types based on the type of researchers who conduct the replication [32, 33]. An internal replication is conducted by original researchers. External replications are conducted by different researchers. External replications are considered as more value since it is conducted by researchers who are without a vested interest and therefore less bias [34]. We conducted an external replication and details are provided in

Chapter 4. In contrast to the original study, we analyse more versions of the studied open source software systems. We also analyse differences between the original study and our replication in detail.

1.4 Thesis contributions

The contributions of this thesis are as follows:

C1: A comprehensive taxonomy of maintenance-related tasks with a catalogue of characteristics of tasks (what do tasks impact, when would they be performed, etc.). These characteristics (e.g., whether a maintenance task impacts code, component or overall system architecture) support to deeper understand of maintenance tasks including the architectural impact of a maintenance task. This taxonomy and catalogue are based on a thorough analysis of the state-of-the art of maintenance tasks. This is useful for researchers in the future to support their research since this will contain synthesized knowledge and pinpoint gaps in the current research landscape. Also, practitioners will get insights about current practices that can help benchmark their own practices and use the catalogue to analyse the potential impact of a maintenance task.

C2: A comprehensive understanding of architecture evolution based on a replication of an empirical study that analyses architectural changes of open source software systems. The replication is conducted externally. It adds credibility to original study by analysing more releases than the original study, including newer releases. Furthermore, the replication enriches the body of software engineering knowledge by providing insights about architecture evolution of different types of releases (major, minor, etc.) with offers insights into how architectures change (when, to which extent, the trends of change over time, etc.). Also, analysing releases over a longer period than the period covered in the original study shows if the trends and extent of architectural change slow down over the lifetime of a system or not. Finally, the replication explores differences (and reasons for differences) between the original study and the replication. This offers additional insights into why findings about architectural change can differ even though the same systems are analysed using the same techniques.

1.5 Overview of the thesis

The remainder of this thesis consists of four chapters:

Chapter 2: ***Background.*** This chapter provides the foundations related to the problem investigated in this thesis and describes the basics related to *software maintenance*, *software architecture* and *architecture decay*.

Chapter 3: ***A Systematic Mapping Study on Software Maintenance Tasks and their Characteristics.*** This chapter reports the results of a systematic mapping study that investigates software maintenance tasks and their characteristics. The mapping study provides answers to RQ1 (see Section 3.6.2 and 3.6.3) of the thesis and provides contribution C1 (see Section 1.4).

Chapter 4: ***Architecture Decay in Open Source Software Systems: an External Replication.*** This chapter presents a replication of an empirical study which was conducted to investigate architecture evolution by quantifying architectural changes across different types of version pairs of open source software systems system. Through this replication we provide answers to RQ2 (see Section 4.8) of the thesis and provide contribution C2 (see Section 1.4).

Chapter 5: ***Conclusions*****.** This is the last chapter of this thesis which summarizes the answers to the research questions raised in Section 1.2, the conclusions drawn, contributions, and discusses future research.

2 Background

This section introduces some background related to software maintenance, software architecture and architecture decay.

2.1 Software maintenance

In today's world, to remain useful, all software systems need to continuously evolve to satisfy user requirements. The usefulness of a software system depends on the functionality and features it provides to its users as well as its quality, e.g., its availability, correctness and performance [4]. In order to facilitate these characteristics, continuous maintenance of the software is required.

In early 1983, the U.S. General Accounting Office defined software maintenance as "all work performed on a software product after it has gone into production" [38]. Similarly, the IEEE Standard 1219-1993 [39] defines maintenance as "the modification of a software product after delivery, to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment." The ISO/IEC 12207 Standard for Life Cycle Processes [40] describes maintenance as "modification to code and associated documentation due to a problem or the need for improvement. The objective is to modify existing software product while preserving its integrity."

Some researchers and practitioners use the term "evolution" as a synonym for maintenance [35]. Since the term "evolution" lacks a standard definition, Rajlich and Bennett [35] introduced a stage model where maintenance and evolution are two distinct stages or phases of a lifecycle of a software system. According to this model, evolution starts immediately after the first release of a system to adapt the software to continuously changing user requirements and operating environment. Furthermore, according to Rajlich and Bennett, evolution implies *substantial* changes (e.g., to change architecture to support adding a new functionality) to the application that lead to a new release or a major version [35]. On the other hand, according to Rajlich and Bennett, maintenance is more about minor changes (e.g., minor code change to improve performance, etc.) and may not result in a new release or version. For example, both evolution and maintenance involve fixing errors, but fixing errors in maintenance is related to bug fixes which lead to a patch release while fixing errors in evolution is a part of continuous functional

enhancements which lead to a major release. Therefore, maintenance tasks can be considered part of evolution.

Software maintenance can be classified into four types: corrective maintenance (fixing bugs), preventive maintenance (aims at improving design), adaptive maintenance (changes related to adding new features) and perfective maintenance (improving quality of the software system) [36, 37]. More details of this classification and other classifications will be discussed in Section 3.2

Furthermore, maintenance and evolution cause a range of *changes* in the application. A software change is the basic operation of both software evolution and maintenance. Regarding change, evolution and maintenance differ based on the difficulty of the change (i.e., allowing substantial changes in evolution leading towards a release and minor changes in maintenance as discussed above) [35].

In this thesis, we use maintenance to refer to general post-delivery activities which include evolution when there are substantial changes related to functionality enhancement. A change will refer to a modification/process or activities involved with maintenance and evolution.

2.2 Software architecture

The discipline and notion of software architecture started to emerge in the late 1960s [38-40]. Since then, the concept and meaning of software architecture evolved. In 1969, the initial notion of architecture as software design or high-level design of a software system was introduced. At this point, architecture was viewed from the perspective of system structure and behavior focusing on technical aspects (i.e., involving components, connectors, design style, patterns, etc.) where architecting was mostly considered as drawings of boxes and lines with different semantics. Until the early 90s, the concept of software architecture was not clearly differentiated from software design [38, 40]. Then, in the 1990s, the concept of software architecture emerged as a distinct discipline and the term “architecture” was used contrast architecture and design [38, 40]. In their seminal paper, Perry and Wolf [38] defined software architecture as “software architecture = elements, form, rationale” which represents a software architecture as a set of architectural elements: elements (components) that have a particular form (constraints and relationships/connectors among components) that are created and combined based on a rational (the preferred architectural style i.e., the choice of elements and form).

In 2000, the IEEE 1471 standard defined software architecture as the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution [37]. In 2003, Bass et al. [41] defined software architecture as the structure of the system which includes software element (e.g., subsystems, layers, packages or components in the sense of component-based software engineering), the externally visible properties of the elements (what the system does, how the system does it) and relationships among the software elements (e.g., how they interact or depend on each other).

More recently, there was a shift towards socio-technical aspects and architecture was considered from a stakeholder's point of view looking at how stakeholders reason and make decisions [42]. ISO/IEC/IEEE 42010 suggests capturing architecture using different views. This means, one system can have multiple architectures or architecture descriptions. Architecture descriptions document an architecture using one or more architecture views. An architecture view or simply "view" addresses one or more concerns of the system stakeholders. *Concerns* are stakeholder's interests that are related to system's environment including developmental, technological, business, organizational, political, economic, legal, ecological and social influences [43]. There can be different types of concerns, such as concerns regarding stakeholder needs, goals, design constraints, quality attributes, etc. Some examples for concerns according to this standard are functionality, usage, security, evolvability, complexity, communication, business goals and customer experience. These concerns are framed as views using viewpoints. A view is then governed by its viewpoint (i.e., a viewpoint can be considered a template for creating a view for a particular system). The viewpoint consists of conventions that construct, interpret and analyse the view such as languages, notations, design rules, modelling methods, views analysis techniques etc.

In 2005, Jansen and Bosch [44] defined architecture as a composition of a set of explicit architectural design decisions. They define an architectural design decision as a description of set of architectural additions, subtractions and modifications to the software architecture, including a rationale (the reason behind design decision or why a change is made to the software architecture), design rules, design constraints and additional requirements that to be satisfied by the architecture as a result of a design decision. *Architectural* design decisions are concerned with decisions that have a global or system-wide (rather than local) impact (e.g., architectural style which has an impact on interactions between components) and decisions that impact quality attributes of a software system. A quality attribute is a characteristic or

feature that affects a system quality, i.e., the degree to which a system meets requirements based on user needs or expectations [45]. There are three basic types of quality attributes: design-time (e.g., maintainability, portability, testability, usability), runtime (e.g., performance, security, availability, functionality, usability) and intrinsic quality attributes (e.g., conceptual integrity, correctness, completeness) [46]. Also, internal implementation details that are local to a component (internal algorithms, data structures etc.) are usually not system-wide and not considered as architectural [41]. Furthermore, architectural decisions are those design decisions that are difficult to make right and hardest to change later [41]. For example, changing architectural pattern (e.g., from client-server to peer-to-peer) which is architectural is costly and difficult than changing an implementation of an algorithm inside a class later. Therefore, any changes related to architecture require a careful analysis and a good understanding before making a change.

In 2006, Kruchten et al. [47] pointed out the importance of architectural knowledge created or managed to build and evolve quality systems. Architectural knowledge consists of architecture design as well as design decisions including the reasoning behind decisions, assumptions, frameworks, reference architecture (i.e., an architecture template for all the software systems in a particular domain that captures the fundamental components of the domain and relationship between these components.) and the other factors that drive those decisions (such as company policies, standards that have to be used, previous experiences of the architect, etc.). Kruchten et al. [47] "updated" the formula of Perry and Wolf as, "architecture knowledge = architecture design + architecture design decisions" [39, 42].

Software architecture is known as the heart of any software and central to map the changes in requirements and their implementation in the source code [48]. An architecture abstracts these implementation-specific details by modelling lines-of-code as architectural components and their relationships. The quality of an architecture determines quality attributes of a system which impact different types of stakeholders [41]. For example, users are concerned with availability, reliability, etc. Developers are concerned with maintainability, scalability etc. Managers are concerned with completing the implementation of a system on time. More importantly, software architecting is about making early design decisions for a system. It is important to avoid or at least reduce potential risks that come with the changes. Jansen and Bosch highlight that the complexity, high costs of change, and architecture decay (see Section 2.3) are some of the major challenges related to software architecture design [44].

2.3 Software architecture decay

Due to continuous maintenance, evolution and time pressure, developers often take shortcuts, perform changes in an ad-hoc manner (careless additions, modifications, removal of architectural design decisions [49]) without analysing the impact of changes on the system architecture (see Section 1.1). These uncontrolled changes cause the architecture moving away from its original intended design. This negatively impacts the ability of a system to accommodate unplanned modifications [2]. Eventually, the architecture becomes significantly different from the intended architecture. As a result, architecture decay occurs, i.e., the system becomes more difficult to understand and to change, and the chances of introducing unexpected bugs increase [2, 10, 12, 50]. In the worst case, such systems are no longer maintainable and required to be reengineered [51].

Software architecture decay is not a new concept and it has been discussed using different terms. In 1992, Perry and Wolf used the term "*architecture erosion"* to describe when the originally designed architecture (conceptual architecture) becomes inaccurate with respect to the implemented architecture [38]. Hochstein and Lindvall [52] refer to "*architecture degeneration"* when the actual (as-is, as-built, as-implemented, or concrete) architecture of a system deviates from the planned (as-designed, ideal, intended or conceptual) architecture. Riaz et al [1] use the term "*architecture decay"* to explain architecture degeneration, but also highlight that in case of decay the architecture no longer satisfies the quality attributes that guided the initial architecture design. van Gurp and Bosch explain "*design erosion"* as design decisions taken at an early stage of system evolution which conflicts with requirements that need to be satisfied later during system evolution [53]. Izurieta and Bieman describe "*design decay"* as the deterioration of the internal structure of system designs [54].

There are several well-known examples of architecture decay. One example is the Netscape web browser. In 1998, shortly after its release, the development of a new version of the Mozilla web browser started. When Mozilla developers started to work with the existing source code, they found out that the original code was too difficult to work with and concluded that code was eroded beyond repair. They decided to re-code Mozilla from scratch. Until, 2002 they were unable to reach version 1.0 of Mozilla. When Godfrey and Lee analysed Mozilla, they concluded that either its architecture has eroded significantly over a short period or its original architecture was not properly designed [52, 53]. Another example is the Linux kernel. One reason it took nearly two years to release version 2.4 after

releasing the previous stable release was that the code of the previous version needed a massive restructuring in order to support new requirements. After redesigning some major parts of the old version, new requirements could be implemented, and the performance was also improved [53].

3 A Systematic Mapping Study on Software Maintenance Tasks and their Characteristics

In this chapter we report the results of a systematic mapping study (SMS) to examine software maintenance tasks and their characteristics and to answer RQ1 of this thesis.

3.1 Introduction

Maintenance is key to any successful and long-living software product. Controlling and handling the changes in a software system is one of the greatest challenge that software engineers face during the period of maintenance and evolution [2, 4]. As argued earlier in Section 1.1, managing and implementing maintenance tasks can be difficult due to various reasons, such as using incorrect/inconsistent terminology, lack of sufficient detail of maintenance tasks, etc. Consequently, practitioners may face difficulties in prioritizing tasks, handling cross-cutting tasks, budgeting, staffing, allocating resources etc. as explained in the problem statement (see Section 1.1). Also, software maintenance consumes a large portion of the resources of a software project over its life cycle as discussed before in Chapter 1.

To overcome above difficulties, a comprehensive understanding of software maintenance is useful. One way of understanding maintenance task is to describe maintenance tasks in detail, including different types of maintenance tasks and characteristics (see RQ1). In order to answer RQ1 we investigated the state-of-research related to software maintenance through a mapping study.

The contribution of this mapping study is two-fold. First, the study provides synthesized knowledge to researchers related to the current state-of-the-research on software maintenance, future directions, and gaps in current research landscape that need further attention. The second contribution is beneficial for practitioners: The study provides a taxonomy of concrete maintenance tasks (providing a common terminology and that supports understanding maintenance tasks in detail) and a catalogue of characteristics that can be used as a framework to characterize maintenance tasks. Furthermore, insights from the literature could help practitioners benchmark their own practices. The benefits of these outcome of our study will be discussed further when introducing the research questions in Section 3.4 .

The rest of this chapter is organized as follows: Section 3.2 describes existing secondary studies related to software maintenance, Section 3.3 describes related work, Section 3.4 introduces the research questions of this mapping study, Section 3.5 elaborates on the mapping study design, Section 3.6 presents the results, Section 3.7 discusses the results further, Section 3.8 presents limitations of our study and threats to validity, and Section 3.9 concludes this chapter.

3.2 Existing secondary studies

Before conducting a full mapping study, we conducted an informal search to find existing secondary studies on software maintenance.

We found two *mapping studies* that are related to software maintenance and evolution. One was published in 2013, focusing on software evolution visualisation [55]. The other one was published in 2012 on aspect-oriented software maintenance [56].

We also found several *systematic literature reviews* (SLR) related to software maintenance. A SLR differs from a mapping study mainly in the type of research questions asked, and analyses conducted as well as the quality assessment of reviewed primary studies. SMSs aim at studying broad research questions about a specific topic while an SLR aims at analysing in depth more specific research questions of the topic (see Section 1.3). Riaz et al. [57] conducted a SLR on software maintenance, focusing on software maintainability prediction and metrics. In 2009, Benestad et al. [58] conducted a SLR covering the period of 1993–2007. This SLR aimed at understanding software maintenance and evolution and classifies changed-based studies based on the goals of the studies and summarized the change attributes (e.g., change count, change effort, change size, code quality, etc.). This study is slightly related to our SMS since it also supports understanding maintenance. However, this study classifies attributes related to maintenance tasks and not the tasks themselves. In 2010, Williams and Carver conducted a SLR focusing on characterizing architectural changes [2].

Based on a series of tertiary studies covering the period from 2004 to 2009 conducted by Kitchenham et al. [59], [60] and a tertiary study by da Silva et al. [61], we found more SLRs that focus on fault prediction [62], defect detection [63], program comprehension [64], code duplication [65] and mining software repositories in the context of software evolution [66].

Since above secondary studies are not helpful for answering our research questions, we conducted our own mapping study. We decided to perform a systematic mapping study over a systematic literature review since a mapping study provides a comprehensive overview of a broader research area (i.e., software maintenance tasks and characteristics in our study) answering broader research questions regarding the current state of the research on the topic.

3.3 Related work

Since RQ1 aims at identifying, classifying and describing characteristics of software maintenance tasks based on the literature, we discuss studies related to classifying and characterizing software maintenance tasks in this section (note that we briefly introduced basic types of maintenance in Section 2.1, but provide a more detailed discussion here). Since there are several types of classifications, below we describe these classifications in more detail.

3.3.1 Classification of maintenance

According to the IEEE 1219-1998 standard [37], software maintenance can be classified into two main types or purposes: correction (*corrective* and *preventive* maintenance) and enhancements (*adaptive* and *perfective* maintenance). *Corrective* maintenance is about fixing defects, such as interface errors, logic errors, syntax errors, etc. *Preventive* maintenance aims at avoiding predictable problems in the future, for example by correcting design or architectural flaws that make software difficult to maintain. Enhancements involve adding new features (*adaptive* maintenance) and improving performance or the quality of the software (*perfective* maintenance).

More than three decades ago, Lientz and Swanson [67] proposed a classification of software maintenance that classifies maintenance into three types: *adaptive* maintenance is applied to properly adapt a system to external environmental changes, *perfective* maintenance is applied to eliminate inefficiencies, enhance performance or to improve maintainability based on user requests, and *corrective* maintenance is about fixing errors in a software.

In 1988, Lin and Gustafson [68] argued that the classification of Lientz and Swanson is based on the intention of the maintainer and not the maintenance activities that actually occur in the software. For example, *adaptive* maintenance in Lientz's and Swanson's classification is about adapting the software based on

environmental changes, and not about what happens in the software to adopt to environmental changes. Lin and Gustafson argue that such classification based on the maintainer's intentions. For example, *corrective* maintenance in Lientz's and Swanson's classification refers to just fixing errors and does not describe what happened to the software (such as modifying/adding statements in code), in contrast to *corrective* maintenance in the classification of Lin and Gustafson which describes in more concrete terms what can happen to the software. Therefore, considering only the maintainer's intensions will not allow managers (who allocate resources, manage staff, manage products, etc.) to understand what is really being done. Therefore, Lin and Gustafson extended the classification of Lientz and Swanson, based on the changes done to the software and introduced six types of maintenance: *corrective* (correcting errors in source code), *adaptive* (adding new functions or deleting functions from the code to meet changes of the requirements), *retrenchment* (temporarily removing a function from executable code, e.g., by commenting out code), *retrieving* (removing comments and "reactivate" code), *prettyprinting* (adding properly formatted comments to increase easy code readability) and *documentation* (adding new comments to explain source code).

In 2000, Chapin et al. [69] proposed another classification of activities that are involved throughout software maintenance and evolution based on the changes that occur in the software. This classification includes 12 types of activities which are grouped into four clusters (support interface, documentation, business rules, software properties). The maintenance type is decided based on three criteria which are in the form of questions (see decision tree in Figure 4). Maintenance types are indicated at the end of each branch in the decision tree shown in Figure 4.

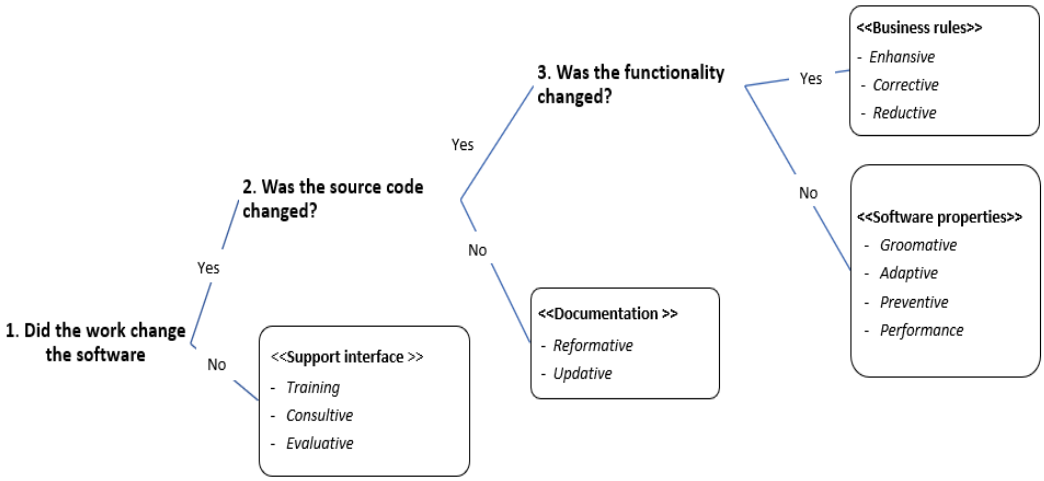


Figure 4: Decision tree proposed by Chapin for classifying maintenance types

1. Support interface cluster: maintenance types related to how systems or technology personnel interact with stakeholders with respect to the software.
    i. *Training*: activities related to training of development team on a software that was changed.
    ii. *Consultive*: activities related to estimating time and cost of changes for proposed maintenance works.
    iii. *Evaluative*: activities related to evaluating a software after changes; includes testing (e.g., regression testing, stress testing, diagnostic testing, etc.), debugging, etc.
2. Documentation cluster: maintenance types related to documentation.
    iv. *Reformative*: activities related to improving the readability of documentation, preparing training materials, etc.
    v. *Updative*: activities to replace obsolete documentation with current documentation, preparing UML models to document existing source code, etc.
3. Software properties cluster: maintenance types related to properties or characteristics of the software.
    vi. *Groomative:* changes to improve maintainability and security (e.g., replacing components or algorithms with more well-designed ones, changing authorization access levels, etc.).
    vii. *Adaptive:* changes to adapt a software to a new environment.
    viii. *Preventive:* modifications to avoid/simplify future maintenance.
    ix. *Performance:* activities that affect the user, but not the functionality of a system, such as replacing algorithms or components with faster ones, etc.
4. Business rules cluster: maintenance types related to business rules and functionalities experienced by the user.
    x. *Enhancive:* adding or replacing business rules to extend or expand the system's functionality.
    xi. *Corrective:* fixing bugs, changing the handling of exceptions, etc.
    xii. *Reductive:* removing or reducing functionality (e.g., removing components/algorithms/subsystems or reducing data flows, etc.)

3.3.2 Classification of change

In addition to literature related to types of software maintenance, there are also classification of software change:

Buckley et al. [70] proposed a taxonomy of software change that focuses on underlying mechanism of change rather than the purpose of the change (i.e., why something is changed) which was the main focus of previous classifications. This classification classifies changes based on characterizing the mechanisms of change and the factors that influence these mechanisms. Software change mechanisms refer to software tools used to achieve software evolution and the algorithms underlying these tools. This taxonomy is based on 15 dimensions that focus more on technical aspects, i.e., the *how*, *when*, *what* and *where* the software changes occur. The dimensions in this classification are:

1. *time of change* (runtime, design-time, etc.)
2. *change history (*sequential or parallel changes*)*
3. *change frequency* (continuously, periodically, or at arbitrary intervals)
4. *anticipation* (were changes foreseen or not)
5. *artefact* (software artefacts that changed, such as requirements, architecture, design, source code, documentation, etc.)
6. *granularity* (granularity of the change, such as system, subsystem, package, class, object, variable, method or statement)
7. *impact* (e.g., local or system-wide changes)
8. *change propagation* (required follow-up changes elsewhere in the system)
9. *availability* (must the system be available when changes are made)
10. *activeness* (proactive: self-adaptive, i.e., system initiates/implements change itself; reactive: system reacts to events/changes driven by external agents)
11. *openness* (is system open to every possible change and specifically built to allow for extensions)
12. *safety* (are safety aspects of a system preserved after change)
13. *degree of automation* (are changes automated, partially automated or manual)
14. *degree of formality* (change mechanism can either be implemented in an ad hoc way or based on some underlying mathematical formalism)
15. *change type* (semantic-preserving, e.g., refactoring activities which change only the structure of code but not system functionalities; semantic-modifying e.g., changes involve modifications to the system functionality.)

In 2018, Elkholy and Elfatatry [71] proposed a software change taxonomy focusing on analysing the impact and cost of software change. They classified changes based on four parameters: change reason (new user requirements and errors that arise at runtime), change level (such as requirement, design and code), change effect

(change impact and change propagation) and change system properties (coupling between components and dependencies related to system functions).

3.4 Mapping study research questions

This mapping study will answer RQ1 of this thesis: ***What is the state-of-the-art of maintenance-related tasks*?** RQ1 is concerned with obtaining a comprehensive understanding of software maintenance tasks. To answer this, we decomposed the RQ1 into two sub-questions:

RQ1.1: **What are the most frequently reported maintenance tasks?**
This RQ extracts concrete maintenance tasks from the literature on software maintenance and classifies them into different types (e.g., bug fixing, refactoring etc.). This provides a comprehensive taxonomy describing maintenance tasks in a hierarchical structure, i.e., it shows how high-level tasks can be split into concrete maintenance tasks. For example, a high-level task such as bug fixing may involve several concrete tasks, e.g., changing a return type of a method, learning about the source code before actually making the change etc. Also, the classification includes properly named tasks that can be used as a common taxonomy that may help reduce inconsistent terminology.

RQ1.2: **How can maintenance tasks be characterized?**
This RQ extracts characteristics of maintenance tasks from the literature. Answering this RQ provides a catalogue of characteristics that can be used as a framework to evaluate and characterise the tasks extracted in RQ1.1. Furthermore, these characteristics may support identifying maintenance tasks that involve architectural changes which will be investigated in more detail in the next chapter of this thesis.

Both above research questions help understand maintenance and overcome difficulties mentioned in the introduction of this mapping study and explained in more detail in the problem statement of the thesis. The concrete maintenance tasks of the taxonomy (RQ 1.1) and their characteristics (RQ 1.2) can be used when allocating resources. Also, the terminology of maintenance tasks used in the taxonomy provides a consistent terminology which can be used when communicating during planning or estimating maintenance tasks. Breaking down tasks into lowest possible level (RQ 1.1) also helps handle cross-cutting tasks (see Section 1.1). Furthermore, based on the characteristics of tasks (RQ 1.2), tasks can be evaluated based on level of impact of change. For example, a maintenance task that has an impact at the architectural level requires more budget since the task is

more complex than a task related to a change inside a method/class. We reflect on the outcome of the research questions and how they address the problems mentioned in the introduction in more detail in Chapter 5.

3.5 Study design

This section presents the protocol of the systematic mapping study. The protocol contains a plan and the procedure used to answer our RQs. Our protocol is presented according to the guidelines proposed by Petersen et al. [30]. The execution of the protocol is shown in Figure 5. The details of the steps shown in the figure are presented in the following subsections.

3.5.1 Search scope

We performed an automatic search on electronic databases to retrieve the relevant studies using the search string introduced in Section 3.5.2. Performing an automatic search is the dominant method for identifying relevant papers in systematic mapping studies [72]. Note that we consider only a selected set of venues (conferences and journals) based on their quality. The list of selected venues is included in Appendix A. This was to ensure the quality of primary studies analysed in this mapping study. Also, we excluded grey literature (i.e. workshop papers, book chapters, and technical reports) which is not peer-reviewed and usually of lower quality compared to papers published in peer-reviewed conferences and journals [72]. Furthermore, we included only empirical, exploratory and descriptive types of research papers since we were interested in what tasks are involved in maintenance, the type of tasks (e.g., classifications), characteristics of tasks, etc. rather than the papers that propose maintenance techniques or tools. We further specified these in inclusion and exclusion criteria, see Section 3.5.3.

The search scope of this mapping study, including the period of search and the electronic sources searched is as follows.

- **Time period**: We searched primary studies published in the last eight years (from January 2010 to November 2017). This was to focus on the most recent studies on maintenance tasks. The end date is November 2017, because this mapping study started in November 2017.
- **Electronic databases**: To broaden our search, we used more than one database. There are several standard databases that are typically used in

mapping studies and we selected databases that are widely used in literature studies in the software engineering [28, 72]:

1. IEEE Xplore
2. ACM Digital Library
3. ScienceDirect
4. Scopus
5. Springer Link

Chen et al. [72] suggested that these databases are easily accessible and provide efficient means to conduct mapping studies. Chen et al. [72] also suggested some other databases like Kluwer Online, Wiley InterScience etc., but we did not choose those databases because they are not as popular in software engineering.

3.5.2 Search strategy and search string

The search strategy needs to include proper search strings to ensure completeness of the coverage of relevant studies. Inappropriate search strings cause too many irrelevant studies or missing many relevant studies [73]. The search string used in this mapping study was defined as follows:

- Initially, a set of key words related to software maintenance was defined based on the title and the research questions of this study. The initial set of keywords included "maintenance", "evolution" and "change". Then, alternative terms or synonyms for selected search terms ("bug fixing", "update" and "correction") were obtained by analysing subject headings used in databases and journals and added to the initial set of search terms. Another set of keywords was added to indicate the type of study we were interested in, i.e., "empirical", "exploratory" and "descriptive".
- The first test search string was constructed by using various combinations of the search terms with the use of Boolean ANDs and ORs. We validated our search string using a quasi-gold standard as proposed by Zhang et al. [74]. A quasi-gold standard for an automated search is a set of known studies relevant to a particular topic which were manually selected from few venues. This set of studies can be used to evaluate the results of an automatic search. The known studies selected according to the quasi-gold standard must also be included as a subset of studies retrieved from the automatic search. Otherwise, the search string used in the automatic search should be revised.

We defined the quasi-gold standard as a set of 13 papers (see Appendix B that were selected manually from several high quality software engineering venues: International Conference on Software Engineering (ICSE), International Conference on Software Maintenance (ICSM), International Conference on Software Maintenance and Evolution (ICSME), Working Conference on Reverse Engineering (WCRE), Working Conference on Mining Software Repositories (MSR), European Conference on Software Maintenance and Reengineering (CSMR) and Journal of Systems and Software.

- The search string was improved iteratively and finalized by comparing it with the quasi-gold standard. The final search string was confirmed since all 13 papers also appeared in the results automatic search.

The finalized search string was as below:

"(maintenance OR evolution OR change OR update OR bug fixing OR correction) AND (software) AND (empirical OR exploratory OR descriptive)"

Based on the options and search settings offered by the databases, the search string was adjusted for each electronic database. The search strings used for each database are shown in Table 1 and were applied on the "abstract" field. In SpringerLink, the results were refined by the selected time intervals and also by sub-disciplines: software engineering, information systems applications, programming languages/compilers/interpreters, programming techniques, software engineering/programming and operating systems, management of computing and information systems, algorithm analysis and problem complexity, user interfaces and human computer interaction, computers and society, IT in business, data mining and knowledge discovery, system performance and evaluation, special purpose and application-based systems, computer science (general), models and principles, systems and data security, computer applications, pattern recognition, software management, e-commerce/e-business, business information systems, complexity, popular computer science, performance and reliability, coding and information theory, quality control/reliability/safety and risk, engineering design and mathematical applications in computer science.

Table 1: Search strings

Database	Search string
IEEE Xplore	("Abstract": maintenance OR "Abstract": evolution OR "Abstract": change OR "Abstract": update OR "Abstract": bug fixing OR "Abstract": correction) AND ("Abstract": software) AND ("Abstract": empirical OR "Abstract": exploratory OR "Abstract": descriptive)
Scopus	ABS ("maintenance" OR "evolution" OR "change" OR "update" OR "bug fixing" OR "correction") AND ABS ("software") AND ABS ("empirical" OR "exploratory" OR "descriptive") AND (LIMIT-TO (SUBJAREA, "COMP")) AND (LIMIT-TO (LANGUAGE, "English"))
ScienceDirect	("Abstract": maintenance OR "Abstract": evolution OR "Abstract": change OR "Abstract": update OR "Abstract": bug fixing OR "Abstract": correction) AND ("Abstract": software) AND ("Abstract": empirical OR "Abstract": exploratory OR "Abstract": descriptive)
ACM	Abstract: ((maintenance, evolution, change, update, bug fixing, correction) AND (software) AND (empirical, exploratory, descriptive))
SpringerLink	(software maintenance OR software evolution OR software change OR software update OR bug fixing OR software correction) AND (software) AND (empirical OR exploratory OR descriptive)

3.5.3 Study selection

We used the following criteria to include and exclude a particular paper in the mapping study. A paper was included if it satisfied all inclusion criteria and was removed if it met at least one exclusion criterion. We had two inclusion criteria:

- I1. The paper refers to the software maintenance tasks.
- I2. The paper presents empirical, exploratory or descriptive work.

The exclusion criteria were:

- E1. The paper is not in English.
- E2. The paper is not in the software engineering domain (e.g., health, construction).
- E3. The paper is not published in the selected set of venues as listed in Appendix A.

- E4. The paper discusses maintenance of computer hardware.
- E5. The paper is not a peer-reviewed publication (i.e., a workshop paper, book chapters that are not conference papers).
- E6. The paper is published in the form of an abstract, tutorial, workshop summary, poster abstract or talk or short paper that does not contain enough details to answer research questions.
- E7. The paper is a duplicate of other papers. We exclude duplicates, such as same publication included in multiple databases, or conference papers followed by a journal article (we only included the journal version [73]).

We followed the selection process shown in Figure 5. We removed duplicates among the papers retrieved from automatic search by applying E7. The set of papers left was filtered further in few rounds as follows. In the first round, we filtered papers based on the quality by only looking at the venue applying E3 and E5. In the second round, we removed papers which were completely out of scope by looking at the title of the paper. If we were not sure about a paper, it was included in the next round of filtering. In third round, we filtered the papers left from the second round further by reading the abstract, and keywords. If we were not sure about a paper, we kept the paper and went to the next round. In the fourth round, we filtered the papers left from the third round by reading the full text and applying inclusion and exclusion criteria. If we were not sure about a paper, researchers discussed a paper to make a final include/exclude decision.

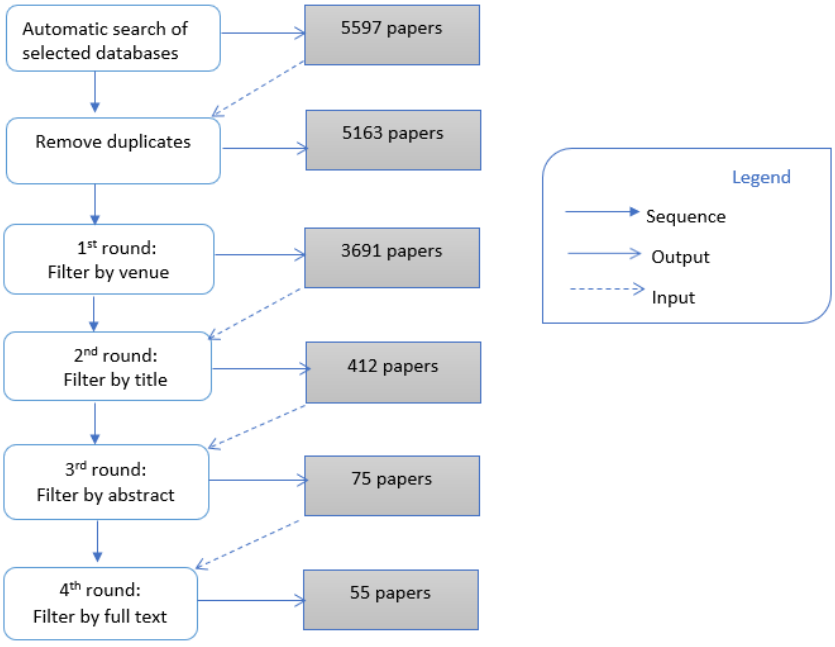


Figure 5: Study selection process

3.5.4 Data extraction and analysis

To answer research questions explained in Section 3.4, we extracted the data items listed in Table 2 from each selected study and recorded data in a spreadsheet.

Table 2: Data items extracted from selected studies

Data item	Description	Relevant RQ
Year	Publication year	None
Venue	Publication venue	None
Publication type	Journal, conference	None
Maintenance tasks	Maintenance task(s) as described in papers	RQ1.1
Characteristics of maintenance tasks	Characteristics of maintenance tasks as described in papers	RQ1.2

For maintenance tasks, we extracted tasks as described in papers. If there was a new task described in a paper, we added the task (and the paper in which it appeared) to the list of already extracted tasks. If a task had been found in a paper before, we added the current paper to the list of papers that describe that task. A similar approach was taken for characteristics of maintenance tasks described in papers. Furthermore, we used descriptive statistics and frequency analysis of extracted data to answer the research questions.

3.6 Results

We obtained 55 papers (see Appendix B) for data extraction and below we present the results.

3.6.1 Overview of selected papers

Figure 6 shows the distribution of the 55 papers over the past 8 years i.e., from 2010 to 2017. Papers were published as a journal paper (21 papers) or a conference paper (34 papers). Furthermore, we plotted the numbers of papers for each year (see Figure 7) which shows the trend of the number of published papers related to software maintenance. We noticed that the number of published papers has been increasing until 2015 but decreasing since 2015.

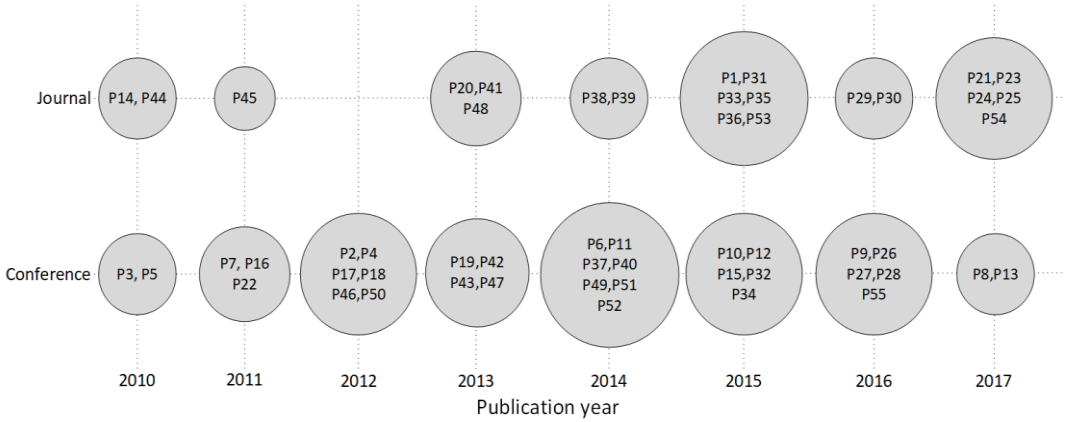


Figure 6: Distribution of papers over publication type and year

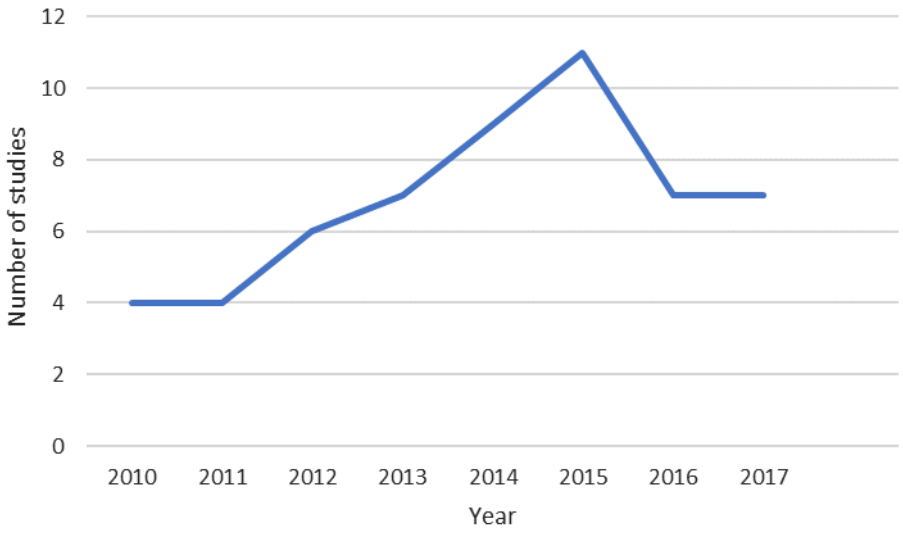


Figure 7: Distribution of papers over time period

3.6.2 Software maintenance tasks (RQ 1.1)

This section provides the answer to RQ 1.1 which is about investigating frequently reported software maintenance tasks.

We collected many types of software maintenance tasks at different levels, from a very abstract level to more concrete levels. The levels emerged from analysing maintenance tasks in papers. Maintenance tasks were merged to form a taxonomy in a hierarchical structure.

3.6.2.1 Overview of taxonomy

The resulting taxonomy classifies maintenance tasks into nine main types, see Figure 8. The taxonomy is presented in ten figures: Figure 8 shows the main types of maintenance as an overview and Figures 9 to 17 show the sub-types of each main type. The papers where tasks have been extracted from are also shown at the end of each branch using their IDs as listed in Appendix B.

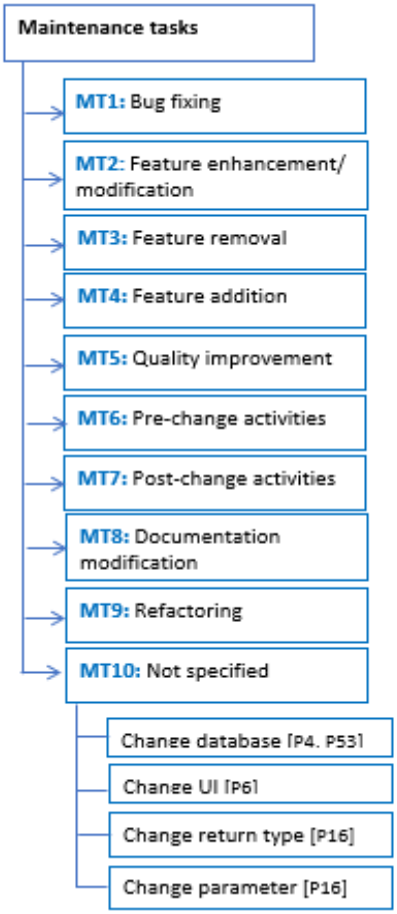


Figure 8: Overview of taxonomy for software maintenance tasks: main types

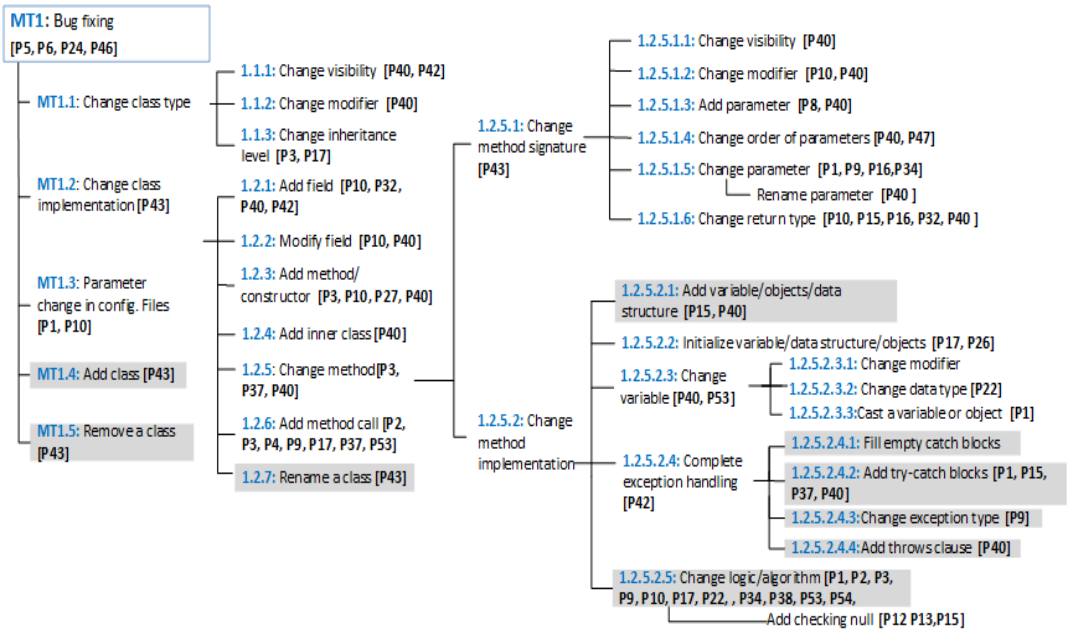


Figure 9: Maintenance type "Bug fixing" (MT1)

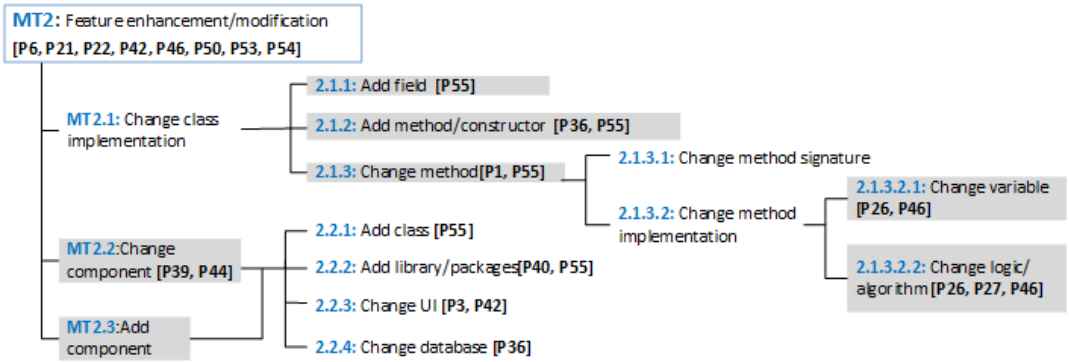


Figure 10: Maintenance type "Feature enhancement/modification" (MT2)

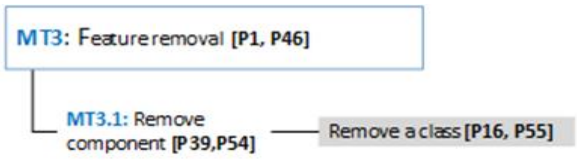


Figure 11: Maintenance type "Feature removal" (MT3)

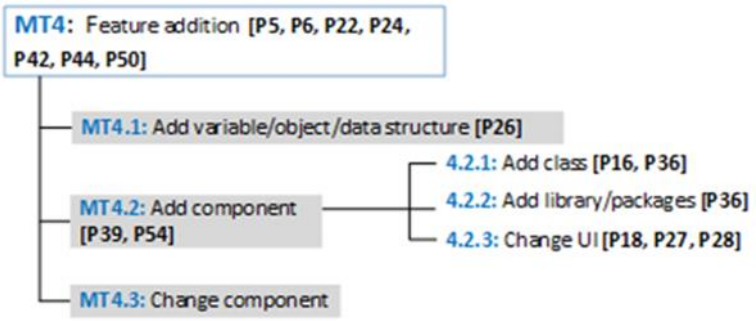


Figure 12: Maintenance type "Feature addition" (MT4)

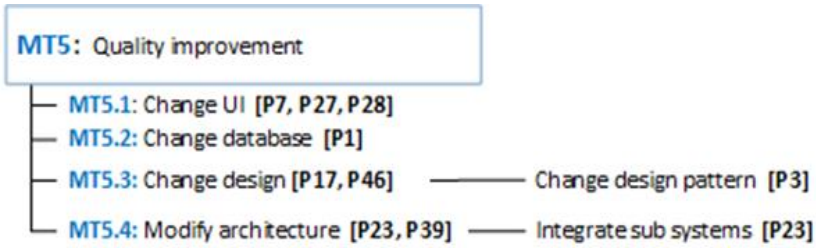


Figure 13: Maintenance type "Quality improvement" (MT5)

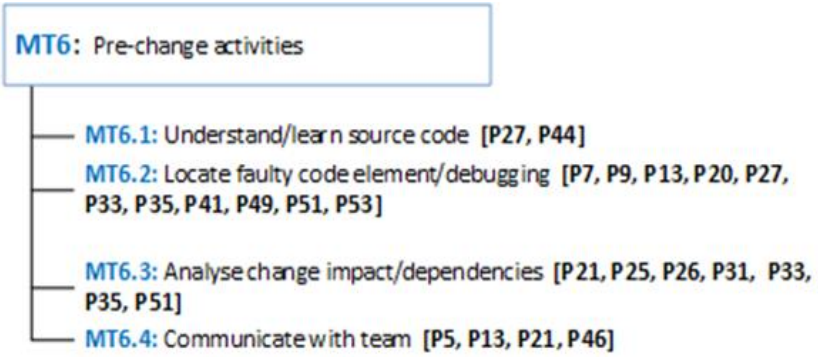


Figure 14: Maintenance type "Pre-change activities" (MT6)

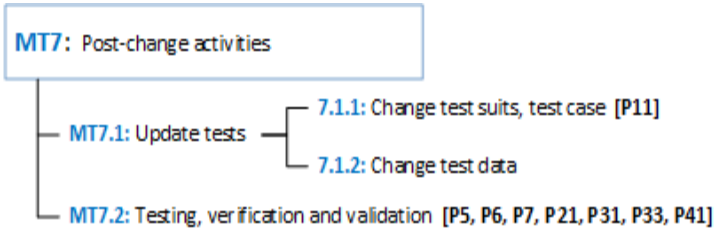


Figure 15: Maintenance type "Post-change activities" (MT7)

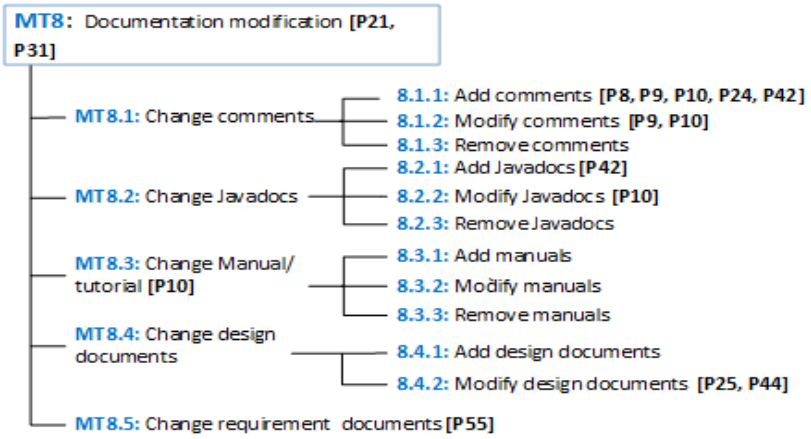


Figure 16: Maintenance type "Documentation modification" (MT8)

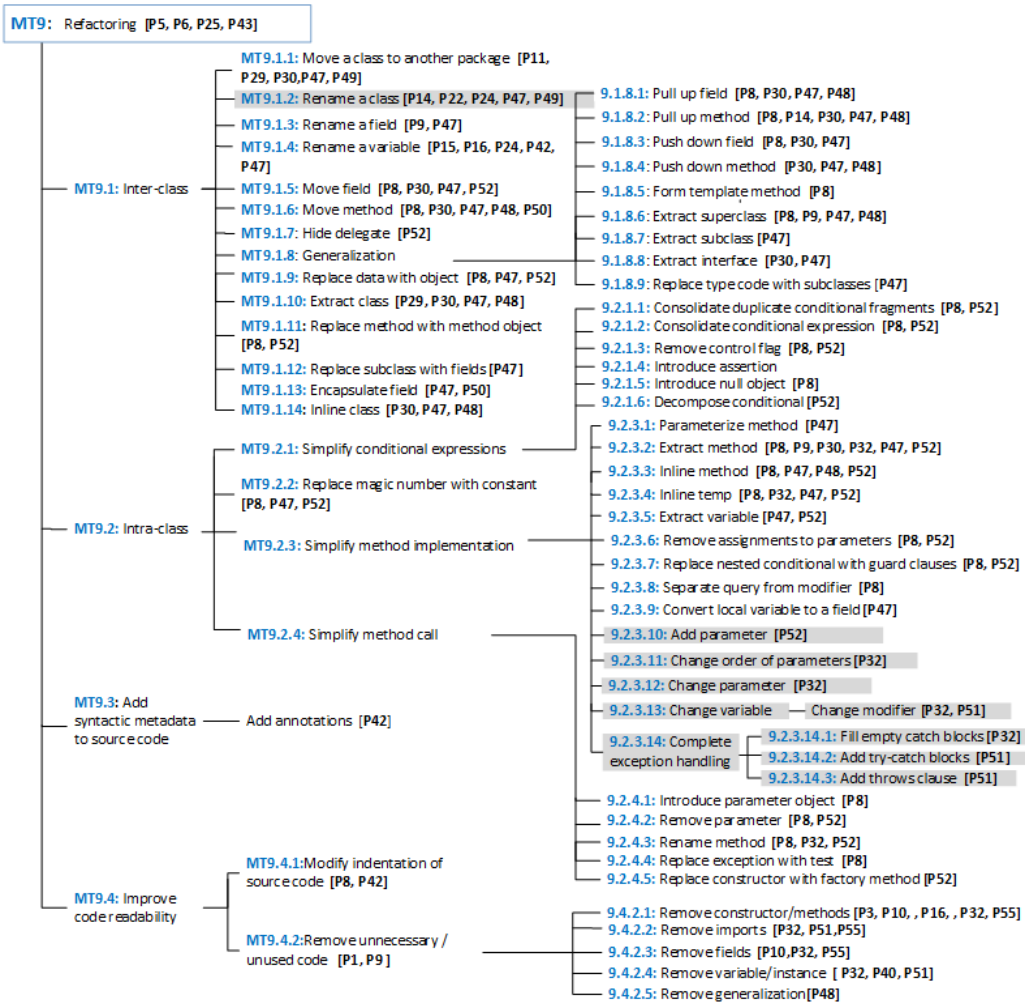


Figure 17: Maintenance type "Refactoring" (MT9)

While classifying papers, we found some tasks that could not be classified easily. One such case is *exception handling* (MT1.2.5.2.4, 9.2.3.14). Some papers consider implementing exception handling as refactoring while other papers consider it as part of bug fixing. Such tasks are shown in the taxonomy with a grey background and appear under multiple maintenance types. Similarly, *change UI* (MT2.2.3, 4.2.3, 5.1) and *change database* (MT2.2.4, 5.2, 10) are performed with different intentions, such as improving quality or adding/enhancing a feature. Since there are many papers for both types, we classified those tasks using separate branches for each type rather than displaying the two types separately in the taxonomy. Finally, papers that state only concrete tasks/sub-tasks without their purposes or intention or any information that would allow us to determine the type of maintenance were classified under the category "Not specified" (see MT10 in Figure 8).

3.6.2.2 Frequently reported types of maintenance tasks

Figure 18 shows the distribution of papers for types of maintenance and publication year. The total of number of papers relevant to each type is indicated at the end of each corresponding gridline of the graph. This shows that the most frequently reported high-level maintenance types are *bug fixing* (MT1) and *refactoring* (MT9). The second most frequently reported maintenance types are *feature addition* (MT4), *enhancement* (MT2), and *pre-change activities* (MT6).

The most frequently reported maintenance types are either *bug fixing* or *refactoring* but there is a drop in these two categories after 2015. Instead, a considerable increase can be seen in *documentation modification* (MT8) and *pre-change activities* (MT6). Furthermore, we noticed that *documentation modification* (MT8) was not often reported or not reported at all in the previous years but gained more attention in the recent years and became the most frequently reported type in 2017. One reason for this could be that support for *bug fixing* (MT1) and *refactoring* (MT9) has matured over the years and less research is necessary.

Furthermore, we noticed that in the last two years (2016 and 2017), most of the papers that are categorized under *bug fixing* and *refactoring* address either *documentation modification* (MT8) or *pre-change activities* (MT6): P13 and P28 explored *pre-change activities* such as root cause tracking of bugs and impact analysis respectively, while other papers use documentation, such as investigating code comments (P8, P9 and P29), technical debt-specific documentation (P25),

model-driven development in the context of maintenance context (P54) and consider documentation as a part of the refactoring process (P30).

The number of papers that reported on activities related to *feature addition* (MT4) and *enhancement/modification* (MT2) was quite stable throughout the reviewed period. Maintenance types such as *feature removal* (MT3), *post-change activities* (MT7) and *quality improvement* (MT5) are the least reported types. Among these types, it is interesting that fewer papers are concerned with *quality improvement* (MT5) of software as a dedicated maintenance activity. The reason for this may be that researchers are interested more in improving the functionality or features of a system rather than improving quality of a system. Another reason could be that all tasks related to *quality improvement* (MT5) are not reported explicitly. For example, some papers consider fixing performance issues, security-related issues, etc. as *bug fixing* (MT1).

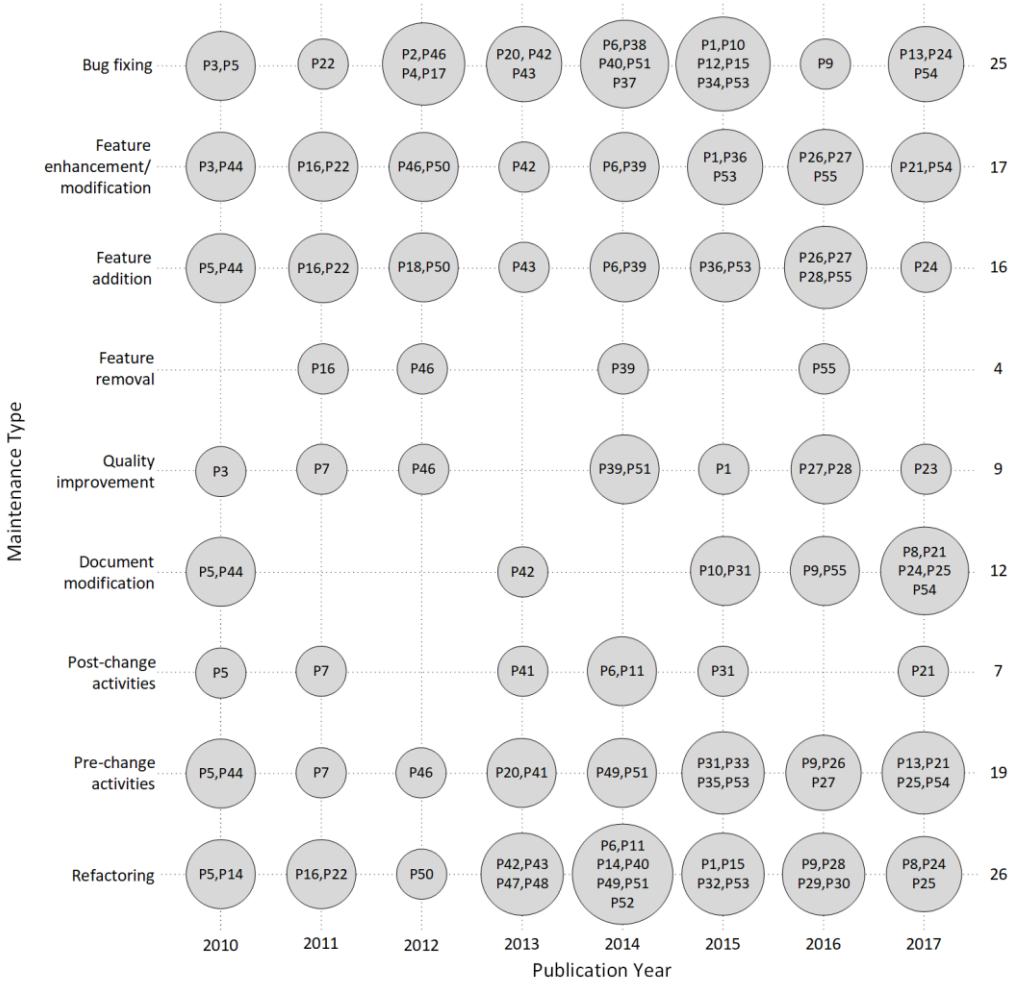


Figure 18: Distribution of studies over maintenance type and publication year

Other papers which are classified under *refactoring* (MT9) also contribute to *quality improvement* (MT5) since refactoring increases the quality of a software system by improving design, readability and reducing bugs. Furthermore, refactoring positively affects quality attributes such as extensibility, modularity, reusability, complexity, maintainability and efficiency of a software [75].

3.6.2.3 Frequently reported maintenance tasks

Table 3 shows the list of concrete tasks that have been reported in more than 5 papers. According to the table, the most frequently reported maintenance task is *modifying an algorithm* (MT1.2.5.2.5, MT2.1.3.2.2) which appears under two higher-level maintenance tasks, i.e., under *bug fixing* (MT1) and *feature enhancement/modification* (MT2). *Modifying an algorithm* includes tasks such as adding break/continue statements (P10), changing data structure (P17), changing mathematical or logical operations (P26, P27). The second most reported maintenance task is locating faulty code element (MT6.2). This also includes tasks which are explained as tracking root causes of bugs (P20, P13), bug localization (P7, P33), debugging (P41) and detecting code smell (P9). There were only few or no studies related to some sub-types of *documentation modification* (MT8), such as change Javadocs (MT8.2), manuals (MT8.3) and design documents (MT8.4).

Table 3: Most frequently reported maintenance tasks

Maintenance type	Maintenance tasks	Frequency
Bug fixing	Add field	5
	Add method/constructor	6
	Add method call	7
	Change parameter	6
	Change return type	5
	Change algorithm	14
	Add try-catch blocks	5
Pre-change activities	Locate faulty code/debugging	11
	Analyse change impact /dependencies	7
Post-change activities	Testing/verification and validation	7
Documentation modification	Add comments	5
Refactoring	Move a class to another package	5
	Rename a class	6
	Rename a variable	5
	Move method	5
	Pull up method	5
	Extract method	6
	Remove constructor/method	5
Feature addition/ enhancement, quality improvement, Not specified	Change UI	7

3.6.3 Characteristics of maintenance tasks (RQ 1.2)

Table 4 presents the catalogue of characteristics that can be used to characterize maintenance tasks that were identified when answering RQ 1.1. The characteristics were grouped into seven main types:

1. Impacted artefact: what type of artefact (requirement, architecture, component, class, method, test, documentation) of the software is impacted by the maintenance task.
2. Target: whether the intention of the task is to maintain functional requirements or quality attributes.
3. Impacted stakeholders: type of stakeholder impacted by a maintenance task.
4. Anticipated complexity: complexity level of the maintenance task based on factors such as number of files and lines changed, expertise required, etc.
5. Timing: whether the task is performed during design/run time or pre/post release.
6. Tool support: whether a maintenance task can supported by automated, semi-automated tools or if it cannot be automated.
7. Frequency: whether a task is performed once or frequently.

Each above type has sub-characteristics. The papers where these sub-characteristics appear and the number of papers are listed in the last two columns of Table 4. The frequency of each main type, i.e., the sum of unique paper IDs of its sub-characteristics is listed next to the main type in the first column. According to the catalogue, the most frequently occurring ways of characterizing maintenance tasks are based on whether maintenance tasks can be supported by tools or not (29 papers), impacted artefacts (28 papers) and the target of the task (22 papers) i.e., functional requirements/quality attributes.

We further analysed the papers that address the characteristic, *tool support* (see Section 3.7.2, Table 9). Among the papers that are classified based on *impacted artefacts*, architectural impact is the mostly addressed sub-type and maintainability under the characteristic target. The least frequent way of characterizing maintenance tasks is based on *impacted stakeholders*, *anticipated complexity*, *timing* and *frequency*.

Table 4: Characteristics of maintenance tasks

Characteristics of maintenance tasks				Frequency
1. Impacted artefacts (28)	• Requirement	P21, P30, P41, P45, P46, P51, P54, P55		8
	• Architecture/Design	P7, P11, P14, P17, P20, P21, P23, P29, P41, P43, P44, P46, P49, P51		14
	• Components/modules	P11, P21, P27, P29, P43, P44		6
	• Classes	P38, P41, P43, P51		4
	• Methods	P8, P14, P30, P38, P51, P52		6
	• Test	P21, P30, P41, P46, P51, P54		6
	• Source file or non-source file (Config. File)?	- Non-source file	P1, P10, P17	3
		- Source file	P1, P10, P17, P30, P36, P41, P46, P51	8
	• Documentation	P21, P30, P31, P36, P41, P42, P48, P51, P54, P55		10
2. Target (22)	• Functional requirement	P1, P21, P26, P 27		4
	• Quality attributes			
	Design-time quality attributes:			
	- Maintainability	P14, P16, P22, P23, P24, P29, P30, P41, P44, P48, P49, P50, P51, P53, P54		15
	✓ Modularity	P32		1
	✓ Reusability	P23, P29, P30, P32, P49		5
	✓ Analysability	P32, P44		2
	✓ Testability	P23, P32, P44		3
	✓ Modifiability	P32, P44		2
	✓ Adaptability	P49, P51		2
	- Understandability	P23, P24, P29, P30, P32, P50, P51		7
	- Changeability	P32, P38, P41		3
	- Flexibility	P16, P23, P29, P30		4
	- Evolvability	P23		1
	- Scalability	P14		1
	- Functionality	P44		1
	Runtime quality attributes:			
	- Performance	P14, P32, P46, P50		4
	- Security	P32		1
	- Reliability	P23, P32, P44		3
	✓ Availability	P16		1
	✓ Fault tolerance/ Robustness	P51		1
3. Impacted stakeholders (4)	• Developers	P1, P44		2
	• Users	P1		1
	• Need Experts	P1, P7, P12		3
	• PM	P44		1
	• Architects	P44		1
	• Testers	P44		1
4. Anticipated complexity (7)	• Complex	P7, P16, P30, P32, P51, P53, P54		7
	• Simple	P54		1
5. Timing (4)	• Design-time	P19, P39, P45		3
	• Runtime	P19, P39		2
	• Pre- release	P5		1
	• Post-release	P5		1
6. Tool support (29)	• Yes	P8, P10, P12, P15, P19, P23, P25, P26, P29, P30, P31, P32, P34, P36, P37, P40, P41, P42, P44, P47, P48, P49, P50, P51, P54, P55		26
	• No	P8, P10, P12, P15, P19, P23, P25, P30, P31, P36, P44, P47, P50		10
	• Semi-automatic	P3, P14, P16, P19, P29, P30, P32, P50, P51, P55		10
7. Frequency (1)	• Once	P12		1
	• Repetitive/Recurring	P12		1

3.7 Discussion

This section provides an analysis and interpretation of the results of this mapping study.

3.7.1 Frequently reported maintenance tasks

We extracted nine maintenance types from the extracted data of selected studies (see Figure 8). In general, *bug fixing* (MT1) and *refactoring* (MT9) are the two types that gained the most attention from the software engineering research community. Only in the last two years, tasks related to *documentation modification* (MT8) and *pre-change activities* (MT6) gained more attention. Among all types of maintenance, the most reported concrete-level maintenance task is *changing algorithm* (MT1.2.5.2.5, 2.1.3.2.2).

Feature removal (MT3), *post-change activities* (MT7) and *quality improvement* (MT5) are the least reported maintenance types. One potential reason could be that these types are not studied explicitly because modifying or enhancing features will automatically remove the old feature and also always involve post-change activities like testing, verification and validation. Some papers related to *quality improvement* (MT5) are classified under *refactoring* (MT9) or *bug fixing* (MT1), when issues related to security and performance are considered as bugs.

3.7.2 Characteristics of maintenance tasks

We extracted seven main types of characteristics (see Table 4) that can be used to characterize a maintenance task. The most reported ways of characterizing a maintenance task are characterizing based on *tool support*, *impacted artefact* (mostly architectural impact) and *target* of the task (mostly maintainability).

In Table 5, Table 6, Table 7 and Table 8, we present a cross-tabulation of maintenance tasks that were discussed when answering RQ1.1 (see Section 3.6.2) and the characteristics that were discussed when answering RQ1.2 (see Section 3.6.3) to identify relationships between the maintenance tasks and characteristics. The first two columns of tables showing cross-tabulations (Table 5, Table 6, Table 7 and Table 8) present types of maintenance and concrete maintenance tasks. The remaining columns present the paper IDs of papers that characterize a maintenance task based on the given characteristic, the number of papers (i.e., sum of unique paper IDs in a row), the total number of papers relevant to a given maintenance task

(regardless of how these papers characterize a task) and the percentage of papers characterized based on the given characteristic (e.g., 60% for "Add field" means that 60% of papers that discuss maintenance task "Add field" characterize that task based on the impacted artefact).

Table 5: Cross-tabulation of maintenance tasks and characteristic 1

Maintenance type	Maintenance tasks	Characteristic 1: Impacted artefacts									No. of papers address the characteristic	No. of papers reported on task	%of papers discuss the characteristic
		Requirement	Architecture/Design	Component	Class	Method	Test	Source-file	Non-source file	Documentation			
Bug fixing	Add field	P55						P10	P10	P42, P55	3	5	60
	Add method/ constructor	P55		P27				P10, P36	P10	P36, P55	4	6	67
	Add method call		P17					P17	P17		1	7	14
	Change parameter							P1	P1		1	6	17
	Change return-type								P10	P10	1	5	20
	Change algorithm	P46, P54	P17, P46	P27		P38	P46, P54	P1, P10, P17	P1, P10, P17, P46	P54	7	14	50
	Complete Exception handling	P51	P51		P51	P51	P51	P1, P51	P1	P42, P51	3	8	38
Pre-change activities	Locate faulty code element	P7, P41, P51	P20, P41, P49, P51	P27	P41, P51	P51	P41, P51	P41, P51		P41, P51	6	11	55
	Analyse change impact	P21, P51	P51	P21	P51	P51	P21	P51		P21, P31, P51	3	7	43
Post-change activities	Testing/verification, validation	P21, P41	P7, P21, P41	P21			P21, P41	P41		P21, P31, P41	4	7	57
Documentation modification		P21, P55	P11, P21, P44	P11, P21, P44		P8	P21	P10	P10	P21, P31, P42, P55	8	11	73
Refactoring	Move a class to another package	P30	P11, P29, P49	P11, P29		P30	P30	P30		P30	4	5	80
	Rename a class		P14, P43, P49	P43		P14					3	6	50
	Rename a variable									P42	1	5	20
	Move method	P30				P8, P30	P30	P30		P30, P48	3	5	60
	Pull up method	P30	P14			P8, P14, P30	P30	P30		P30, P48	4	5	80
	Extract method	P30				P8, P30, P52	P30	P30		P30	3	6	50
	Remove method/ constructor	P55						P10	P10	P55	2	5	40
Feature addition/ enhancement, quality improvement, Not specified	Change UI		P7	P27						P42	3	7	43

Table 6: Cross-tabulation of maintenance tasks and characteristic 2

Maintenance type	Maintenance tasks	Characteristic 2: Quality attributes		No. of papers address the characteristic	No. of papers reported on task	%of papers discuss the characteristic
		Design-time	Runtime			
Bug fixing	Add field	P32(6)	P32(3)	1	5	20
	Add method/constructor			0	6	0
	Add method call	P53		1	7	14
	Change parameter	P16(2), P32(7)	P16, P32(3)	2	6	33
	Change return-type	P16(2), P32(7)	P16, P32(3)	2	5	40
	Change algorithm	P22, P38, P53, P54	P46	5	14	36
	Complete Exception handling	P32, P51(3)	P32, P51	2	8	25
Pre-change activities	Locate faulty code element	P41(2), P49(3), P51(3), P53	P51	4	11	36
	Analyse change impact	P51(3)	P51	1	7	14
Post-change activities	Testing/verification/validation	P41(2)		1	7	14
Documentation modification		P24 (2), P44 (5)	P44	2	11	18
Refactoring	Move a class to another package	P29(4), P30 (4), P49(2)		3	5	60
	Rename a class	P14(2), P22, P24 (2), P49(2)	P14	4	6	67
	Rename a variable	P16(2), P24 (2)	P16	2	5	40
	Move method	P30(4), P48, P50(2)	P50	3	5	60
	Pull up method	P14, P30(4), P48	P14	3	5	60
	Extract method	P30(4), P32(6)	P32(3)	2	6	33
	Remove method/ constructor	P16(2), P32(6)	P16, P32(3)	2	5	40
Feature addition/ enhancement, quality improvement.	Change UI				7	0

Table 7: Cross-tabulation of maintenance tasks and characteristic 3

Maintenance type	Maintenance tasks	Characteristic 3: Complexity	No. of papers address the characteristic	No. of papers reported on task	%of papers discuss the characteristic
Bug fixing	Add field	P32	1	5	20
	Add method/constructor		0	6	0
	Add method call	P53	1	7	14
	Change parameter	P16, P32	2	6	33
	Change return-type	P16, P32	2	5	40
	Change algorithm	P53, P54	2	14	14
	Complete Exception handling	P32, P51	2	8	25
Pre-change activities	Locate faulty code element	P7, P51, P53	3	11	27
	Analyse change impact	P51	1	7	14
Post-change activities	Testing/verification/validation	P7	1	7	14
Documentation modification			0	11	0
Refactoring	Move a class to another package	P30	1	5	20
	Rename a class		0	6	0
	Rename a variable	P16	1	5	20
	Move method	P30	1	5	20
	Pull up method	P30	1	5	20
	Extract method	P30, P32	2	6	33
	Remove method/ constructor	P16, P32	2	5	40
Feature addition/enhancement, quality improvement	Change UI	P7	1	7	14

Table 8: Cross-tabulation of maintenance tasks and characteristic 4

Maintenance type	Maintenance tasks	Characteristic 4: Tool support			No. of papers address the characteristic	No. of papers reported on task	%of papers discuss the characteristic
		Yes	No	Semi-automatic			
Bug fixing	Add field	P10, P32, P42, P55	P10, P40	P32, P55	5	5	100
	Add method/ constructor	P10, P36, P40, P55	P10, P36, P55	P3	5	6	83
	Add method call	P37		P3	2	7	29
	Change parameter	P32, P34, P40		P16, P32	4	6	67
	Change return-type	P10, P15, P32, P40	P10, P15	P16, P32	5	5	100
	Change algorithm	P10, P12, P15, P26, P34, P54	P10, P12, P15		6	14	43
	Complete Exception handling	P15, P32, P37, P40, P42, P51	P15	P32, P51	6	8	75
Pre-change activities	Locate faulty code element	P41, P49, P51		P51	3	11	27
	Analyse change impact	P25, P26, P31, P41, P51	P25, P31	P51	5	7	71
Post-change activities	Testing/verification/ validation	P31, P41	P31		2	7	29
Documentation modification		P31, P8, P10, P25, P42, P44, P55	P8, P10, P25, P31, P44	P55	7	11	64
Refactoring	Move a class to another package	P29, P30, P47, P49	P30, P47	P29, P30	4	5	80
	Rename a class	P47, P49	P47	P14	3	6	50
	Rename a variable	P15, P42, P47	P15, P47	P16	4	5	80
	Move method	P8, P30, P47, P48, P50	P8, P30, P47, P50	P30, P50	5	5	100
	Pull up method	P8, P30, P47, P48	P8, P30, P47	P14, P30	5	5	100
	Extract method	P8, P30, P32, P47	P8, P30, P47	P30, P32	4	6	67
	Remove method/ constructor	P10, P32, P55	P10, P55	P3, P16, P32, P55	5	5	100
Feature addition/ enhancement, quality improvement, Not specified	Change UI	P42		P3	2	7	29

These tables show frequently reported maintenance tasks (i.e., tasks that appeared in more than five papers as listed in Table 3) and the four most occurring characteristics. There are two tasks that were considered at a higher-level than the detailed-level considered in Table 3: Instead of "Add try-catch blocks" (MT 1.2.5.2.4.2, 9.2.3.14.2) in Table 3, the next higher-level of this task i.e., *complete exception handling* (MT 1.2.5.2.4, 9.2.3.14) and instead of "Add comments" (MT8.1.1) in Table 3 the next higher-level of the task i.e., *documentation modification* (MT8) were considered in the cross-tabulation, to provide more meaningful results. Below we explain findings with respective to each characteristic based on Table 5, 6, 7 and 8.

Characteristic 1: Concerning the characteristic in Table 5, i.e., characterizing based on *impacted artefacts*, we observed that 80% of the papers that discuss refactoring (MT9) tasks such as *move a class to another package* (MT9.1.1) and *pull up method* (MT9.1.8.2) characterize these tasks based on impacted artefacts. *Add method call* (MT1.2.6), *change parameter* (MT1.2.5.1.5, MT9.2.3.12), *change return-type* (MT1.2.5.1.6) and *rename a variable* (MT9.1.4) are the least characterized based on *impacted artefact*.

Characteristic 2: Concerning the characteristic in Table 6, i.e., *target* quality attributes, the most frequently characterized tasks in terms of this characteristic are related to *refactoring* (MT9). In particular, more than 60% of the papers that discuss *move a class to another package* (MT9.1.1), *rename a class* (MT9.1.2, 1.2.7), *move method* (MT9.1.6) and *pull up method* (MT9.1.8.2) discuss this characteristic. The least addressed tasks in terms of this characteristics are *add method/constructor* (MT1.2.3, 2.1.2), *add method call* (MT1.2.6), *post-change activities* (MT7), *documentation modification* (MT8) and *change UI* (MT2.2.3, 5.1). Only 0-14% of the papers reported on these tasks discuss this characteristic.

Characteristic 3: Concerning the characteristic in Table 7, i.e., *complexity*, we observe that maintenance tasks seem to be rarely characterized based on this characteristic. The tasks most often characterized based on their complexity are *change return-type* (MT1.2.5.1.6), *change parameter* (MT1.2.5.1.5), *extract method* (MT9.2.3.2) and *remove method/constructor* (MT9.4.2.1). *Add method/constructor* (MT1.2.3, 2.1.2), *documentation modification* (MT8) and *rename a class* (MT1.2.7, 9.1.2) are not characterized (0%) based on their complexity.

Characteristic 4: Concerning the characteristic in Table 8 i.e., whether the tasks are supported by tools or not, we note that all the papers that discuss *add field* (MT1.2.1, 2.1.1), *change return-type* (MT1.2.5.1.6), *move method* (MT9.1.6), *pull up method* (MT9.1.8.2) and *remove constructor/method* (MT9.4.2.1) discuss tool support for these tasks. Tasks *add method call* (MT1.2.6), *locate faulty code element/debugging* (MT6.2), *post-change activities* (MT7) and *change UI* (MT2.2.3, 5.1) are tasks that are least characterized by tool support provided for them (only 27-29% of papers related to these tasks address tool support). This may be because these tasks have enough tool support or not necessarily in need of tool support.

We further analysed the papers that address tool support. Table 9 shows the papers addressing tool support with respective to their maintenance type. The papers that are presented by underlined paper IDs proposed tool support. Other papers used tool

support as part of their research or generally discuss existing tool support related to a particular maintenance type.

Table 9: Tool support for software maintenance tasks

Maintenance type	Paper address existing tool support	Frequency
Refactoring	P30, P32, P47, P48, P49, P50, P51, P8	8
Bug fixing	P3, P10, P15, P34, P40, P42, P12	7
Pre-change activities	P26, P37, P40, P48, P49, P29, P41	7
Quality improvement (track and detect technical debt, code and design smell)	P14, P49, P50, P25, P44, P29	6
Documentation modification	P55, P36, P54	3
Feature addition/enhancement	P16, P34	2
Post-change activities (testing)	P31	1

According to Table 9, tool support related to *refactoring* (MT9), *bug fixing* (MT1) and *pre-change activities* (MT6) have received the most attention. This is as expected because these types of maintenance are the most popular maintenance types (see Section 3.6.2.2). Fewer papers report tool support for *documentation modification* (MT8), feature *addition/modification* (MT2,4) and *post-change activities* (MT7).

Based on the results of cross-tabulation, we found that the three tasks *add method* (MT1.2.3, 2.1.2), *add method call* (MT1.2.6), *post change activities* (MT7), *documentation modification* (MT8) and *change UI* (MT2.2.3, 5.1) are not characterized in very detail (see Characteristic 1, 2, 3 and 4). Tasks such as *move class to another package* (MT9.1.1), *move method* (MT9.1.6) and *pull up method* (MT9.1.8.2) are well characterized (compared to other tasks), since the papers related to these tasks often address the characteristics 1, 2 and 4. At least 60% of the papers addressing these three tasks address each of this characteristic (see the last columns of Table 5, 6 and 8). Among above characteristics the characteristic 3 *complexity* is the least used characteristic to characterize maintenance tasks (only up to 40% of the papers related to any task address this characteristic, see Table 7).

3.7.3 Comparison to existing classification of software maintenance

In Section 3.3, we presented existing classifications and types of maintenance tasks. The taxonomy (see Section 3.6.2.1) that emerged from our mapping study is partially reflected in previous works. As mentioned in related work section, Lientz and Swanson [67] introduced three types of maintenance activities considering a maintainer's intention. The all three types that Lientz and Swanson introduced are also included in our taxonomy, but using a different terminology. They used the

terms *adaptive*, *corrective* and *perfective* which refer to feature addition/enhancement, bug fixing and quality improvement in our taxonomy. However, our taxonomy includes more types than Lientz and Swanson s' taxonomy since our taxonomy is based on the different types of tasks developers have to perform. Therefore, the proposed taxonomy also includes types such as documentation modification, pre-change activities, post-change activities and removing features.

The classification of maintenance tasks by Lin and Gustafson [68] introduced six types based on what is actually being done to the software. They consider activities related to improving quality and fixing bug as one type i.e., corrective activities which were considered as two separate categories in our taxonomy. Furthermore, Lin and Gustafson introduced two categories for removing features and documentation similar to our taxonomy. However, the type for documentation-related tasks is not fully consistent with our type of documentation, since we considered it at a broader-level considering all documents types such as code comments, Javadocs, manuals, etc. In contrast, since Lin and Gustafson consider only what happened to the software, only code comments were considered as documentation. Furthermore, activities such as pre/post-change activities, documentation related to design, requirement, user manuals, etc. are not considered in their work.

More closely related to our work is the work of Chapin et al. [76]. They identified 12 types of maintenance activities which are clustered in 4 clusters. These 12 types are closely consistent with the types in our works. They classify documentation as a separate cluster introducing two different types of documentation: *reformative and updative.* These two types consider only the changes that are not related to source code documentation, such as user manuals and design documents. They classified changing code comments as a different type (*groomative*) in a different cluster which is related to improving maintainability and security of a software (software properties cluster) considering code comments as a code grooming activity. Our taxonomy classifies all these types of documentation changes including code comments as one type. Furthermore, they also identified pre and post-change activities as types of maintenance similar to our classification of maintenance types. They separated activities related to testing/debugging, etc. into three separate categories *evaluative*, *training* and *consultive* activities. In our taxonomy, activities related to these three types were classified as pre-change and post-change activities.

Compared to the classification in IEEE 1219-1998 standard, the main difference in our classification is that our work includes several additional maintenance types. For example, we classified documentation related work, pre-change activities such as understanding/learning source code, change impact analysis, discussion with team members, etc., post-change activities such as testing etc. and removing a feature as different types of maintenance.

Considering software change taxonomies, Buckley et al. [70] classified software change activities based on mechanisms of change. The types in their classification are closer to the types in our second classification i.e., catalogue of characteristics of maintenance tasks. All types we identified in this catalogue were also included in their taxonomy. Some types use different terminologies but follow the same definition. For example, *time of change*, *change frequency*, *artefacts* and *degree of automation* in their work refer to what we explained as *timing*, *frequency*, *impacted artefacts* and *tool support* in our catalogue. Furthermore Buckley et al. [70], included another type as *impact* which explains whether the change is local or system-wide. We covered this by including method, class, component, architecture etc. under *impacted artefact* which supports identifying whether a change is local (e.g., in a class) or system-wide (in the architecture). However, their work included types which were not included in our work mainly because our catalogue is based on characterizing concrete maintenance tasks that were discussed in the literature, not based on change mechanism. For example, the category *change history* as in their taxonomy explains whether tasks are performed sequentially or in parallel was not identified in our work as a type of characteristic.

The change taxonomy by Elkholy and Elfatatry [71] focuses on the impact of change to predict the complexity of the change. It considers work done after implementing a change. Elkholy and Elfatatry also discussed the characteristics in our catalogue of characteristics of maintenance tasks. However, the characteristics *tool support* and *frequency* were not discussed in their taxonomy since they focus only the factors that support predicting the impact and complexity of change.

3.7.4 Implications for researchers and practitioners

Concerning **researchers**, we suggest some future directions for maintenance research. Based on the findings presented in Figure 18, we identified that *post-change activities* (validation, verification, updating test, etc.) and *quality improvement* (change UI, database, architecture/design) is less addressed. Furthermore, based on the findings presented in cross-tabulation, we found that

some maintenance tasks are less frequently characterized in detail. In particular, *changing UI* is not well characterized regarding its impact on artefacts, complexity, tool support, etc. Also, only few studies were found that address tool support for maintenance types such as *documentation modification, feature addition/modification* and *post-change activities*. We noticed that even though *documentation modification* (such as updating code comments, Javadocs and documents related to requirement and design) more frequently appeared in research papers in the recent years (see Section 3.6.2.2) still less tool support is provided for this maintenance type. This has also been shown in a mapping study conducted by Zhi et al. [77] who showed that research related to the quality, cost and benefits of documentation is far from mature. Researchers need to further support these areas which are a concern for maintenance but seem neglected in the literature.

Concerning **practitioners**, we recommend that they pay more attention to architectural impact when performing maintenance tasks. Based on the results presented in Table 4, maintenance tasks often seem to involve architectural changes. Therefore, practitioners should be careful not to violate architectural design decisions or introduce technical/architectural debt during software maintenance. Based on the cross-tabulation presented in Table 5, Table 6, Table 7 and Table 8, practitioners can obtain an early understanding of potential impacts of the type of task they perform. Being aware of these characteristics may provide insights about potential consequences of maintenance activities. Furthermore, there are many tools already available for *bug fixing* and *refactoring*. Practitioners can use these tools to make software maintenance easier.

3.8 Threats to validity

The first threat to validity concerns the study selection process employed. The selection of digital libraries, the search string and researcher bias may have resulted in search results that miss some relevant papers. To mitigate this threat, first we selected the most popular digital libraries that index a large number of journals and conferences in the field of software engineering. Then we developed the search string iteratively while performing a trial search on selected databases. Based on trial search, the search string was refined and validated against a quasi-gold standard before performing the automatic search.

However, there can still be papers that we missed that include maintenance tasks which do not explicitly use the search terms we have used but instead using the terms such as “defect”, “refactoring”, “smells”, “design”, “architecture” etc. Such

missing papers may have impacted our results two ways: First, they may have impacted the structure of the hierarchies (i.e., the taxonomy of maintenance tasks and catalogue of characteristics). A missed paper could include tasks or characteristics that might introduce new category (i.e., a new branch) in the hierarchies. Second, it could have impacted the analysis derived from classifying the papers. A missed paper could impact the frequency of papers classified under different categories which might impact the answers to research questions. The impact on the structure and completeness of the taxonomy of maintenance tasks is not significant since adding new papers did not result in new categories in the taxonomy.

To mitigate researcher bias, the selection of papers was reviewed by a second researcher. Initially, the study protocol and inclusion and exclusion criteria was designed and discussed between the researchers. Then, papers were filtered by the researchers and results were compared and discussed. The study protocol is explicitly explained in Section 3.5 in a way that can be followed by other researchers.

Furthermore, we might have been biased during the process of data extraction. This may have resulted in inaccurate data and therefore impacted the classification and analysis of selected studies. To mitigate this threat, we first discussed the data to be extracted (see Table 2) and reached consensus on each data item to ensure a shared understanding of what a data item means. Also, one researcher checked the results of the other researcher to reduce potential inaccuracies of extracted data.

Bias during data synthesis is also another threat which may impact classification of extracted data. Not all papers contain sufficient information about the data item extracted. When it was difficult to properly place a task in the hierarchy, we classified them based on the overall goal (whether a task supports *bug fixing*, *quality improvement*, etc.). When a paper discussed maintenance tasks generally and the intension of a task (e.g., whether *change UI* is due to *a feature addition* or *quality improvement*) is not described, such papers were classified under "Not specified". Furthermore, maintenance tasks related to renaming (class, field, variable and method) were classified under refactoring based on Fowler's definition of refactoring [78] (considering that refactoring does not alter the external behavior of the code yet improves its internal structure) when the type of the maintenance (whether bug fixing or refactoring) was not clearly described in a paper.

3.9 Conclusions

We conducted a systematic mapping study to investigate the state-of-research related to software maintenance to answer RQ1 (*What is the state-of-the-art of maintenance-related tasks?*) of this thesis. In this mapping study, we searched for relevant studies in five main electronic databases and 55 papers were selected for data extraction. Based on the extracted data, we provided a comprehensive understanding of software maintenance tasks as well as an overview of the state-of-the-art of software maintenance. Below, we summarize the conclusions drawn from this mapping study.

1. The results of this mapping study contained a comprehensive taxonomy of concrete maintenance tasks and a catalogue of characteristics that can be used to characterize maintenance tasks. Both the taxonomy and the catalogue organized the tasks and characteristics in a hierarchical structure by grouping them. The papers fall into each tasks and characteristics are also classified in respective hierarchical structures.
2. The taxonomy of maintenance tasks classified maintenance tasks into nine main types: *bug fixing*, *feature enhancement or modification*, *feature removal*, *feature addition*, *quality improvement*, *pre-change activities*, *post-change activities*, *documentation modification* and *refactoring*. The most frequently reported maintenance types are *bug fixing* and *refactoring*. The second most frequently reported maintenance types are *feature addition, enhancement* and *pre-change activities*. Maintenance types such as *feature removal*, *post-change activities* and *quality improvement* are the least reported types. All these types were further classified into sub-types which include concrete maintenance tasks. Among these concrete tasks, the most frequently reported maintenance task is *changing an algorithm* (or logic) in a method's implementation.
3. The catalogue of characteristics group characteristics into seven main types: characterizing based on *impacted artefact* (e.g., component, architecture, requirement etc.), *target* (functional requirements or quality attributes), *impacted stakeholders*, *anticipated complexity*, *timing* (e.g., design-time, runtime), *tool support* and *frequency* (unique or frequent). Each of these types has sub-types. According to the catalogue, the characteristics that were found to describe maintenance tasks most are characterizing maintenance tasks based on *tool support*, *impacted artefacts* and *target*. Only few maintenance tasks are characterized based on *impacted stakeholders* (4 out

of 55 papers), *anticipated complexity* (7 out of 55 papers), *timing* (4 out of 55 papers) and *frequency* (1 paper).

4. Considering the cross tabulation of frequently reported maintenance tasks and characteristics, the tasks *add method*, *add method call*, *post-change activities*, *documentation modification* and *change UI* are not characterized in detail. Refactoring such as *move class to another package*, *move method* and *pull up method* are characterised most compared to other maintenance tasks. In particular characterizing based on impacted artefact, target of the tasks (i.e., quality attributes) and tool support were discussed most (at least in 60% of the papers that report on these tasks).
5. Tool support related to *refactoring*, *bug fixing*, and *pre-change activities* gained the most attention. Fewer papers report tool support to *improve quality*, *post-change activities* and also for *documentation modification* even though documentation in the context of software maintenance has been discussed more and more in the last two years.

4 Architecture Decay in Open Source Software Systems: an External Replication

In the previous chapter, we studied software maintenance tasks as reported in the literature. According to the catalogue of characteristics of maintenance tasks, we noticed that the most frequently occurring ways of characterizing maintenance tasks is based on whether maintenance tasks can be supported by tools or not, the artefacts which maintenance tasks impact, and the target of the task (e.g., functionality or quality attributes). Impacted artefacts and target are directly related to architectural changes: Architectural impact is reported as the most frequent way of characterizing maintenance tasks based on impacted artefact, and quality attributes (target) are achieved through architecture design, as explained in Chapter 1 and 2. This indicates that architectural changes play a major role in software maintenance activities. Therefore, in this chapter we provide insights into architecture evolution by empirically analysing architectural changes to support software maintenance in the context of architecture evolution. In detail, this chapter presents the results of an external replication of a study which empirically assessed architecture evolution of open source software systems.

4.1 Introduction

As discussed earlier (see Section 1.1 and Section 2.3) , the software architecture of a system evolves throughout its lifetime [2, 79]. Architecture decay, also known as architecture erosion or degeneration, is often the result of poor software evolution practices. Decay means that an architecture deviates from the originally intended design and accumulates architectural debt. This makes the architecture more difficult to understand and therefore more difficult to maintain. A good understanding of the nature of architecture decay and architectural change can help developers identify and track decay across the life time of a system.

As explain in Chapter 1, we still lack empirical data of how, where and to which extent architecture changes happen (see RQ2 of this thesis introduced in Chapter 1). Such empirical evidences could help developers make more precise decisions about maintenance and evolution. By addressing RQ2 of this thesis, this chapter contributes empirically grounded insights to understand architecture evolution and decay and to verify how systems evolve throughout their lifetime. In order to

answer RQ2, we conduct a replication of a large empirical study by Le et al. [27] that aimed at understanding architectural change. While other studies (e.g., [50, 80-82]) analyse evolution only at system implementation level (e.g., changes in source code), we analyse architecture evolution. As in the original study, we investigate changes at both system and component level. Analysing architecture change at component-level may expose hidden lower-level changes not visible at system-level but which impact architecture stability. Furthermore, we analyse architecture change between different types of releases of a software system (e.g., between major and minor releases). This replication follows guidelines proposed by Carver [83] and details about the design of the replication are discussed below (see Section 4.7).

There are several contributions of this replication. First, it adds credibility to the original study by analysing 238 additional versions of open source software systems that were not include in the original study, including newer releases of the systems included in the original study. A single study rarely provides absolute answers while replications can add credibility [84, 85]. Second, it enriches the body of software engineering knowledge providing insights about architecture evolution of different types of releases (major, minor, etc.). Analysing releases over a longer period of time i.e., from 02/2000 to 11/2017 (rather than 01/2001 to 06/2014 as in the original study) will show if the trends and extent of architectural change slow down the further a system progresses in its life time. Third, it compares the extent of architecture changes between the original study and the replication and analyses the trends of changes between the two studies. We also explore differences (and reasons for differences) between the original study and the replication. This offers additional insights into why findings about architectural change can differ even though the same systems are analysed using the same analysis techniques.

The rest of this chapter is organized as follows: In Section 4.2, we discuss replications in software engineering in general. In Section 4.3 we discuss the background of our replication and related work. We provide information about the original study in Section 4.4, Section 4.5 and Section 4.6, and introduce the details of the replication in Section 4.7. We then compare the results of the original study with our replication in Section 4.8 and discuss our findings in more detail in Section 4.9. In Section 4.10 we elaborate on validity threats and conclude in Section 4.11.

4.2 Replications in software engineering

Replicating empirical studies is an essential activity in the construction of knowledge in any empirical science [86]. In general, replications are beneficial from both a scientific and an industrial perspective: 1) From a scientific perspective, replications contribute to maturing the body of software engineering knowledge by confirming, refuting or strengthening conclusions drawn from research [32, 87]. 2) From an industrial perspective, replications provide evidence to support practitioners in their decision making [32, 87] For example, demonstrating that a tool or technique gives promising results for different types of projects helps decide in which type of project the tool or technique is effective. This provides an understanding of most appropriate technique practitioners can integrate in their current practices.

There are two types of replications: internal replications (i.e., performed by the researchers who conducted the original study) and external replications (performed by independent researchers) [33]. Carver et al. [88] argue that replications are not a regular practice in empirical software engineering and we need more external replications by independent researchers. External replications help validate results from previous studies and mitigate potential bias of the original researchers. Therefore, external replications offer deeper conclusions and more generalizable knowledge [34, 88, 89].

The lack of external replications has also been discussed by Sjoberg et al. [85] in a survey of controlled experiments published from 1993 to 2002 which found that out of 20 replicated experiments only seven external replications. da Silva et al. [86] showed in a systematic review that the majority of replications between 1994 and 2010 were internal. An extended mapping study updating the results of da Silva et al. [86] showed that nearly 38% of the replications from 2011 to 2012 were external.

In 2015, a mapping study conducted by de Magalhães et al. [90] investigated studies from 1996 to 2013 that reported research *about* replications (i.e., not covering studies that actually report replications). The study classifies papers about replications into eight categories based on the theme or topic addressed: recommendations, replication types, processes, frameworks, tools, guidelines, result combinations and miscellaneous (studies that do not fall in previous categories). The study found that the number of studies about replications is small considering the importance and breadth of topics related to replication work. Magalhães et al. [90] also argue that even though studies classified in the same category (e.g., framework,

tools, etc.) they almost never address the same research problem. This makes it difficult to integrate results of the studies *about* replications and build knowledge to support the design and execution of replications in software engineering. Furthermore, they argue that most of the conducted replications are without studying background about replications (60% of the replications do not cite studies about replications). Even replications that cite studies about replications rarely use suggestions of the cited studies (this indicates that studies about replications which could improve replications have no impact on how replications are conducted).

In 2016, Siegmund et al. [91] argued for the general field of software engineering that replication studies proved useful in other sciences and should be considered more in software engineering. They also found disagreements amongst researchers about the difference between the original study and the replication in order to be considered useful. Also, they found that most researchers appreciate replications but believe they are hard to conduct and publish.

More recent reviews also indicate that replications in the field of software architecture are almost non-existent [92]

4.3 Background and related work

Below we provide an overview of topics relevant to identifying and measuring architecture decay, including architecture recovery and architecture change metrics. Furthermore, we discuss replication in software engineering.

4.3.1 Architecture recovery

As mentioned earlier, software decays over time and architecture descriptions (documented architectures) may not be available or outdated [93]. Even though it is important to keep architecture descriptions up-to-date, this is difficult, costly and adds additional workload. Also, as the software grows in size, it is not easy for developers to remember all the relevant knowledge related to a system's evolution. In this situation, it becomes necessary to extract an architecture from source code. Architecture recovery (often referred to as architecture reconstruction, extraction, discovery, mining or reverse architecting) is the process of extracting an architecture from the implementation (e.g., source code, binaries) of an existing software system. Based on the recovered architecture, we can then analyse decay and erosion by comparing the recovered architectures of two versions of a system

(e.g., before and after a maintenance activity). There are many studies that provide a detailed overview of software architecture recovery techniques as described below.

A survey conducted in 2017 by Zahid et al. [94] provides an overview of architecture recovery techniques that have been proposed from 1998 to 2016. The study categorized recovery techniques into four types:

1. Weight-based clustering groups architectural elements by first assigning weights to entities such as source files, classes, methods, etc. usually based on the number of dependencies between entities and then utilizing a clustering algorithm to group entities to maximize an objective function. For example, Modularization Quality (MQ) introduced by Mancoridis et al. [95] is a well-studied objective function [96-98] in that context.
2. Knowledge-based clustering groups elements using machine-learning approaches.
3. Mapping-based recovery extracts component and connector-based architectures from source code and maps them to logical architectures which consists of high-level design, architectural styles, patterns etc.
4. Program slicing groups architecture elements based on program slicing to extract architectural components and descriptions.

In summary, this study found that most recovery techniques only consider architectural elements (components and connectors), but few consider other architectural information, e.g., architectural styles. Also, the study suggested more work is required to assess the quality of the recovered architecture.

In 2013, Garcia et al. [11] analysed state-of-the-art of architecture recovery techniques. They compared and assessed the accuracy of six recovery techniques: ACDC (Algorithm for Comprehension-Driven Clustering), ARC (Architecture Recovery using Concerns), LIMBO (scaLable InforMation BOttleneck), ZBR (Zone-Based recovery), WCA (Weighted Combined Algorithm) and Bunch. All these techniques were previously published and are supported by tools. The analysis concluded that ACDC and ARC outperform the other techniques. As Garcia et al. [11] argue, most of the existing recovery techniques involve grouping implementation-level entities (source files, classes, or methods) into clusters. Each cluster could then be interpreted as an architectural component. The study also discussed three types of architecture recovery techniques:

1. Dependency-pattern-based: These techniques identify a particular pattern of dependencies: a) structural dependency patterns (structural inputs) such

as control-flow-based and/or dataflow-based dependencies between implementation-level entities (e.g., dependencies among source files based on function calls, access of variables or global variables used in different classes), b) conceptual dependency patterns (textual inputs) such as the words found in source code or comments in implementation-level entities (e.g., dependencies among source files based on file authorship, directory paths).

2. Objective function-based: These techniques partition system entities and their dependencies into components to maximize an objective function (e.g., Modularization Quality mentioned above). These techniques start with a random partition of implementation-level entities and continue until no better (with regards to the objective function) partition can be found or until the objective function is maximized. For example, the *Bunch* recovery technique uses an objective (optimization) function called *Modularization Quality (MQ)* representing the quality of the recovered architecture. To find a partition (to group software entities into clusters) that maximizes MQ, *Bunch* uses hill-climbing and genetic algorithms [11, 93].
3. Hierarchical clustering techniques: These techniques use similarity measures to identify similar entities and form clusters iteratively by grouping similar entities together. The techniques usually start by placing each implementation-level entity in its own cluster. Then, they compute the similarity for each possible pair of clusters based on similarity between entities according to some selected similarity (e.g., Jaccard coefficient, Simple matching coefficient, Unbiased Ellenberg etc.) or distance measures (e.g., Euclidean Distance, Canberra Distance, Minkowski Distance etc.) [99]. Next, the two most similar clusters are combined into one new cluster which then forms a component. This process iterates until all the entities are clustered or the required number of clusters (as decided in advance) are generated [11, 99].

In 2009, Ducasse and Pollet [79] presented an overview of the state-of-the-art software architecture reconstruction approaches and grouped approaches around the goals, process, inputs, techniques and output of the approaches. Similar to Zahid et al. [94], this study found that most approaches consider only limited architecture information (e.g., only components and connectors).

In 2006, Koschke [100] summarized techniques and methods for software architecture reconstruction. The study suggested that future research must address the gap between architecture reconstruction and forward engineering, the lack of

cost models for architecture reconstruction and reconstructing architectures of highly dynamic systems. The study also suggested to embed architecture reconstruction in normal software development processes as regular activities to reduce erosion early on and continuously.

4.3.2 Architecture change metrics

The recovered architectures from different versions of software systems can be used to measure extent of architecture evolution/change or decay among versions. Evolution of an architecture involves activities such as adding, removing or modifying architectural design decisions [27]. The evolution of an architecture can therefore be measured based on some architecture change metrics (similarity/distance metrics) [27]. However, there are only few metrics for quantifying architectural change [27]. Some previous studies related to architecture change metrics include Raibulet and Masciadri [101] who suggested MaAC (Minimum architectural Adaptive Cost) to measure architectural growth in adaptive systems. This metric determines the minimum number of components and connectors which should be added to convert a non-adaptive system into an adaptive system. Nakamura and Basili [102] define a distance metric based on architectures represented by graphs. Due to lack of metrics to measure architectural changes, three new metrics have been introduced in the original study by Le et al. [27] that we replicate in this chapter. We introduce these metrics in more detail in Section 4.4.3.

4.3.3 Replications of studies on architecture evolution

There are only very few external replications related to architecture evolution: Reimanis et al. [103] replicated a case study to investigate potential architecture decay during system evolution by detecting and locating modularity violations. This replication showed that the same tool used in the original study to identify modularity violation in Java systems can also be used for commercial C++ systems. Furthermore, the replication suggested that the applied process for analysing modularity violations did not identify any faults that developers were not aware of. Also, the replication found that developers accept some instances of coupling even though high coupling is considered bad design.

Rossi and Russo [104] replicated a study to understand the evolution of design patterns. It performed the same analysis using the same software systems as in the original study but included newer versions of the studied systems. The results in the original study and the baseline were same when results for versions used in both

studies were compared, but slightly differed for newer versions (the same happened in our study as we will discuss later). For example, the most frequently changing design pattern is different for newer versions of some systems and the reason for change is also different in each study. Furthermore, the replication provided several new insights: 1) For both old and newer (mature) releases the design patterns that change most are patterns that play an important role for the intent of the software. For example, the most frequently changing pattern in JHotDraw (i.e., a Java app for drawing 2D graphics) is "composite" for newer releases (composite supports advanced features such as composing existing figures to provide new features); 2) Changes in the implementation of methods is the type of change that design patterns undergo the most (other types of changes include adding or removing methods, adding or removing subclasses).

An internal extended study [9] of the study that we replicate in this chapter analysed 23 open source systems (including the systems from the original study and an additional 9 systems; 720 versions) and confirmed findings of the original study. However, replicating the study under different conditions by other researchers is necessary to rely on the results of the original study. Therefore, we expanded the original study with more versions (810), including newer releases of the systems used in the original study.

4.4 Original Study

This section provides an overview of the original study published by Le et al. [27] including research questions of the original study, architecture recovery techniques and change metrics, tools used in that study and the study design followed.

4.4.1 Research questions of original study

The research questions of the original study (referred to as OSRQs in this section to differentiate them from the thesis research questions) were as follows:

OSRQ1: *To what extent do architectures change at the system-level?* System-level changes involve additions, removals or moves of implementation entities between components and additions or removals of components themselves. This research question focuses on such changes in a given system's architecture during its development and evolution with respect to when, how and to what extent additions, removals and moves happen.

OSRQ2: *To what extent do architectures change at the component-level?* At component-level, architectural change reflects to which extent two components of an architecture are similar or matching. Whether two components (one from architecture A1 and the other from architecture A2) are different after evolving A1 to A2 is decided based on the number of overlapping implementation-level entities between those components. This question studies the evolution of components with respect to when, how and to what extent these changes happen.

OSRQ3: *Do architectural changes at the system and component levels occur concurrently?* This research question focuses on a comparison between system-level changes and component-level changes with respect to when, why and to which extent changes to the overall architecture are also reflected in the changes to individual system components. It investigates the extent of differences between changes at system and component level and the differences between the general trends at both levels.

OSRQ4: *Does significant architectural change occur between minor system versions within a single major version?* In general, developers introduce a new major version (rather than a minor version) if there are significant changes in the next release which also imply substantial architectural changes (e.g., changes in the API which may cause incompatibility issues between the new and the previous API). This research question investigates if substantial changes occur even though the new version of a system is a minor version within a major version (i.e., minor versions exist in the evolution path before releasing the next immediate major version).

4.4.2 Architecture recovery techniques

The original study used three architecture recovery techniques:

1. **ARC :** Architecture Recovery using Concerns (ARC) by Garcia et al. [105] is a concern-based technique, where a concern represents a concept, role, responsibility or purpose of a software system. ARC leverages system concerns to automatically identify both components and connectors. ARC first extracts concerns from a system implementation (i.e., source code) using information retrieval techniques, i.e., probabilistic topic models and then combines them with structural information (connectors). Topic modelling which is a machine-learning-based approach is used to extract concerns or meaning of source code (in machine-learning, topic modelling

algorithms are used to discover the themes from texts or thematic information such as the main themes, how the themes are connected to each other, etc. from a large collection of documents by analysing words) [106]. For example, ARC recovers concerns from words in the source code (e.g., identifiers and comments) using *Latent Dirichlet Allocation* (LDA) [11, 105]. LDA is a statistical language model which is an example of topic model and used to compute similarity measures for concerns and to identify the concerns present in an individual implementation entity such as a class, function, package, etc. [27, 105]. Furthermore, LDA can represent concerns in a human-readable form. For example, when using LDA, the software system is represented as a set of documents. A document represents an implementation entity, i.e., a class in this work. A document can have several topics, which are considered as concerns in this work. Extracted concerns are of two types: application-specific (i.e., those are related to components, their functionality and their data) and application-independent (i.e., related to connectors and interaction/integration services). Then, similar concerns are grouped together to recover components and connectors based on a similarity measure using a clustering technique. Similarity measures take both concerns and structural information into account. An example for grouping similar concerns is grouping the entities that handle user interface behaviour in one cluster.

2. **ACDC:** Algorithm for Comprehension-Driven Clustering (ACDC) by Tzerpos and Holt [107] consists of two stages. At the first stage, a decomposition of subsystems is constructed where subsystems are identified by grouping implementation-level entities (i.e., functions and variables) of a system into clusters, based on patterns of static dependencies. These patterns group together 1) the entities that reside in the same source file (source file pattern), 2) the source files (e.g., A.h and A.c files in C programing) which contain declaration (interface) and implementation of the same software module (body-header pattern), 3) sets of files (files are considered leafs when a system is represented as a graph) that do not dependent on each other but help achieve a common goal, for example, a set of driver files for peripheral devices (leaf collection pattern) [11, 27, 107], 4) the functions that are accessed by the majority of subsystems (support library pattern), and 5) the entities that belong to a subgraph in the system's graph where the subgraph must contain one node called "dominator node" and a set of other nodes called "dominated nodes" in which any dominated node must go through the dominator node to get to any other node (subgraph dominator pattern). The second stage of ACDC is *Orphan Adoption*, a technique which deals with

evolution of a system. This technique places a newly introduced entity (orphan) in the most suitable cluster [107].

3. **PKG:** Package structure recovery (PKG) is a tool implemented by the authors of original study to extract the package structure of a software system. While ARC and ACDC use algorithms to cluster implementation entities into architectural components, PKG directly extracts package structures from source code by grouping entities that belong to same parent package in the same component [9]. The recovered architectures using PKG directly reflect the package structure of a software system.

4.4.3 Architecture change metrics

Three architecture change metrics have been introduced in the original study to measure architectural changes:

1. *Architecture-to-architecture similarity* (*a2a*): *a2a* is a system-level metric which represents the similarity between two architectures. System-level change refers to the addition, deletion or modification of components. A component represents a cluster which consists of implementation-level entities (e.g., source files in ACDC, classes in ARC), depending on the architecture recovery technique (see Section 4.3.1). The similarity between two architectures (*a2a*) is measured based on the distance between two architectures i.e., minimum number of operations needed to transform one architecture into another. The higher *a2a*, the more similar two architectures are.

   To calculate the minimum number of operations required to transform one architecture to another, Le et al. [27] introduce another distance metric, i.e., *Minimum-transform-operation* (*mto*) which represents the minimum number of operations needed to transform an architecture A1 to an architecture A2 (A1 and A2 are architectures of different versions of the same system). There are five minimum operations represented by *mto*: additions (addE), removals (remE), and moves (movE) of implementation-level entities from one cluster (i.e., component) to another, as well as additions (addC) and removals (remC) of clusters themselves. Note that each addition and removal of an implementation-level entity must follow two operations: To add an entity, it is first added to the architecture and only after that moved to the appropriate cluster (i.e., component); similarly, to remove an entity it is first removed from its cluster (the component that entity belongs to or resided in) and only

then removed from the architecture. The reason for this, as Le et al. [27] explain, is that a recovered architecture is a set of constituent building blocks (i.e., clusters and entities) and their configurations (i.e., arrangement of entities inside clusters). Therefore, changing architectural configuration and changing constituent building blocks are two different operations. Adding/removing entities requires both changing configuration and building blocks. For example, adding an entity will first increase the number of entities in the architecture (one operation i.e., change in the building blocks) and then change one cluster (arrangement of entities inside the cluster, i.e., the change in architectural configuration). This notion behind removing or adding entities is supported by several foundational studies about architecture [108, 109].

The equation to calculate *a2a* is as follows [27]:

$$a2a(A_i, A_j) = 1 - \frac{mto(A_i, A_j)}{\text{aco}(A_i) + \text{aco}(A_j)} \times 100\% \quad (1)$$

$$mto(A_i, A_j) = remC(A_i, A_j) + addC(A_i, A_j) + remE(A_i, A_j), addE(A_i, A_j) + movE(A_i, A_j)$$

$$aco(A_i) = addC(A_\emptyset, A_i) + addE(A_\emptyset, A_i) + movE(A_\emptyset, A_i)$$

where $mto(A_i, A_j)$ denotes the minimum number of operations required to transform architecture A_i to A_j. $aco(A_i)$ denotes the number of operations required to build architecture A_i from scratch when there is no architecture ($A_\emptyset$).

2. *Cluster-to-cluster similarity* (*c2c*): *c2c* is a component-level metric which represents the similarity between two architecture components. At component-level, architectural changes refer to adding or removing implementation-level entities inside architectural components (clusters). *c2c* is measured as the percentage of overlapping implementation-level entities between two clusters. In other words, *c2c* describes whether two clusters can be considered as similar clusters, i.e., the higher *c2c*, the higher the similarity between two components (i.e., clusters).

The equation to calculate *c2c* is as follows [27]:

$$c2c(c_i, c_j) = \frac{|entities(c_i) \cap entities(c_j)|}{\max(|entities(c_i)|, |entities(c_j)|)} \times 100\% \quad (2)$$

where *entities(c_i)* and *entities(c_j)* are the set of entities in clusters c_i and c_j and c_i and c_j are a cluster from version *i* and *j* of system *S*. The number of entities overlapping in the numerator is normalized by the denominator which gives

the number of entities in the larger cluster from the two clusters. This ensures that *c2c* gives the most conservative value for the similarity between two clusters.

3. *Cluster coverage* (*cvg*): *cvg* determines the extent that clusters (i.e., components) of two architectures overlap according to the *c2c* metric. If *c2c* between two components of two architectures is higher than a threshold set by those who analyse the architecture (e.g., 67% in the original study), the two components are considered similar. In other words, *cvg* determines to which extent certain components existed in a previous version of a system or were added in a later version. For example, if A1 and A2 are architectures of version 1 and version 2 of a system, *cvg* (A1, A2) = 70% means that 70% of the components in A1 still exists in A2. *cvg* (A2, A1) = 40% means that 60% (i.e., 100%-40%) of the components in A2 have been newly added.

   The equation to calculate *cvg* is calculated as follows [27]:

$$cvg(A_1, A_2) = \frac{|simC(A_1, A_2)|}{|allC(A_1)|} \times 100\% \quad (3)$$

$$simC(A_1, A_2) = \{c_i | (c_i \in A_1, \exists C_j \in A_2)(c2c(c_i, c_j) > th_{cvg})\}$$

   To compute the percentage of newly added components in A_2, it is required to calculate the number of components in A_1 that have at least one *similar* component in A_2 (i.e., components that have overlapping entities). This is indicated by$|simC(A_1, A_2)|$, which returns the subset of components from A_1 that have at least one similar component in A_2. In detail, $simC(A_1, A_2)$ returns A_1's components that have a *c2c* value above a threshold value (th_{cvg}) for one or more components from A_2. $allC(A_1)$ returns the set of all clusters in A_1. Therefore, the values of *cvg* rely on the values returned by *c2c*.

4.4.4 Original study design and execution

The original study included 14 open source Apache systems with a total of 572 versions. All systems were Java-based and managed in Apache's Jira repository.

Tool used in study: ARCADE (Architecture Recovery, Change, And Decay Evaluator) is a workbench developed by the authors of the original study and provides 1) a suite of ten different architecture recovery techniques, and 2) a set of metrics (see Section 4.4.3) for measuring architecture evolution. Figure 19 shows the workflow of ARCADE and its components and artefacts relevant to this work

(the original study used only parts of the ARCADE workbench, i.e., those parts responsible for architecture recovery and calculating architectural changes, but not components of ARCADE that evaluate decay, quantify symptoms of decay, etc.).

Data collection: The *Source code* (see Figure 19) for all 512 versions of the subject systems was downloaded from Jira. The *Recovery techniques* component (see Figure 19) of ARCADE recovered architectures for each version in the evolution path of each system based on .jar files using three architecture recovery techniques (ARC, ACDC and PKG). According to the implementation of the *Recovery techniques* component (see Figure 19), ARC and ACDC first generate dependency files describing dependencies or relationships between entities. These dependencies are created by analysing dependency patterns (in ACDC) or concerns (in ARC) in source code. For Java-based system, source files are not required since dependencies can be extracted from compiled classes (i.e., the .class or .jar files) as followed in this study. Then, based on these dependency files, clustered files i.e., *Architectures* (see Figure 19) are generated describing ARC, ACDC and PKG architectural views. The dependency files generated from ACDC or ARC are used as inputs to PKG to obtain PKG architectural views (package structures). The generated cluster files, i.e., recovered architectures, are referred to as cluster files and also as architectural files in the study since the content of files form the basis of the architectural views. These cluster files describe architectural views as entity-cluster relationships. Each line in a cluster file represents an entity-cluster relationship where one item in the line represents an entity and the other item the cluster that entity belongs to. For example, below are two lines from the cluster file (.rsf) created with ACDC for ActiveMQ (according to the ACDC algorithm, "ss" denotes subsystems [107]):

```
contain org.apache.activemq.bugs.ss org.apache.activemq.bugs.Receiver
contain org.apache.activemq.bugs.ss java.lang.Exception
```

Metrics calculation: After architectures were recovered from all the versions, the *Change metrics calculator* component of ARCADE (see Figure 19) reads each line of the cluster files, separates the clusters and entities and obtains the count of clusters and entities. Considering the first line from the example above, "`org.apache.activemq.bugs.ss`" is a cluster and "`org.apache.activemq.bugs.Receiver`" is an entity (the second token of text with the suffix "ss" is always a cluster and third set is always an entity). Then, this set of clusters and entities is compared with the set of clusters and entities of another release to calculate the number of added, removed or moved entities or clusters which are required to compute metrics. Based on this information, *Change metrics*

values (see Figure 19) i.e., values for *a2a*, *c2c* and *cvg* are computed for each possible version pair of a system. Then, these metrics values were used to analyse architectural changes as discussed in Section 4.8.

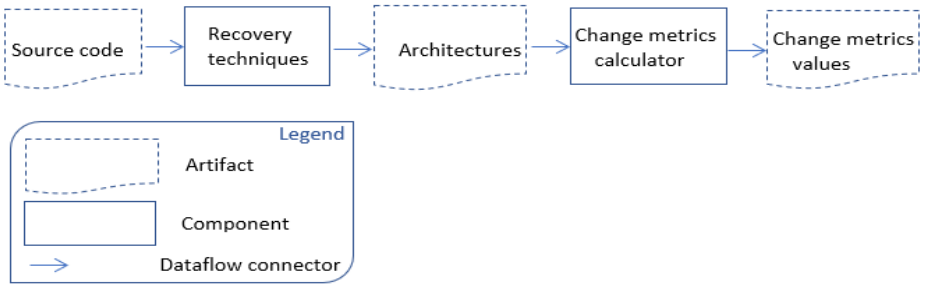


Figure 19: ARACDE's workflow related to this study [27]

4.5 Types of version pairs

The authors of the original study identified four types of versions (*Major*, *Minor*, *Patch*, *Pre*) in a systems' evolution paths and introduced five types of version pairs based on those types. Major and minor versions appear in three types of evolution paths (*Major*, *Minor* and *MinMaj*):

1. *Major*: A major version includes extensive changes to a system's functionality, such as modifications to an API which is not backward-compatible. The evolution path from the start of one major version to the start of the immediate next major version is a "Major" version pair (e.g., version pair [1.0.0, 2.0.0]).
2. *Minor*: A minor version involves smaller changes than a major version where normally backward-compatibility of APIs after the change is ensured (e.g., version pairs [1.0.0, 1.1.0] or [1.1.0, 1.2.0]).
3. *MinMaj:* This type of version pair includes the last minor (or patch) version within a major version and the next major version (e.g., version pair [1.9.0, 2.0.0] when there are no other versions between the versions).
4. *Patch*: A patch version involves smaller changes than minor versions, such as bug fixes or improvements to the functionality of a system (e.g., the version pairs [1.1.0, 1.1.1]), [1.2.0, 1.2.1] and [1.2.1, 1.2.2]).
5. *Pre*: A pre-release version contains new features to get user feedback before releasing the official major or minor version. Pre-releases can be classified as alpha, beta or release candidate (RC). Example pairs are [1.0.0-beta1, 1.0.0-beta2] and [1.0.0-beta2, 1.0.0].

To explain these types further, consider the following versions of the evolution path of a system: 1.0.0, 1.1.0, 1.1.1, 1.2.0, 1.2.1, 1.2.2, 2.0.0-beta1, 2.0.0-beta2, and 2.0.0. In this example, there is only one version pair of type *Major* (i.e., pair [1.0.0, 2.0.0]) in the path. For *MinMaj*, the pair [1.2.0, 2.0.0] appears. The step from 1.2.2 to 2.0.0 can be considered as *MinMaj* if no version 1.2.0 appears in the evolution path. The *Minor* evolution path contains pairs [1.0.0, 1.1.0] and [1.1.0, 1.2.0]. If version 1.1.0 does not in the evolution path, then pair [1.1.1, 1.2.0] can be considered as *Minor*. The *Patch* type pairs consist of the pairs [1.1.0, 1.1.1], [1.2.0, 1.2.1] and [1.2.1, 1.2.2]. The *Pre*-release type pairs contain [2.0.0-beta1, 2.0.0-beta2] and [2.0.0-beta2, 2.0.0].

4.6 Major findings of original study

Below we summarize the findings of original study:

- OSRQ1 (system-level changes) and OSRQ2 (component-level changes): System and component level architecture changes follow the same overall trend: *Major >MinMaj> Minor> Pre> Patch*, i.e., the amount of changes grows from *Patch* to *Major*. Substantial changes are likely to occur also between the end of one major version and the start of the next major version.
- OSRQ3: While architectural changes follow the same overall trend at both system and component levels, the extent of that change differs. In brief, there can be considerable differences between *cvg* values and *a2a* values for the same type of version pair. This difference grows from *Patch* and *Pre*-release type of versions (nearly no difference) to *Major* versions (notable difference between *cvg* and *a2a* values). Furthermore, even though the overall architecture of a system may seem stable, there can be architectural changes within the components. Relying only on stability at system-level may hide "lower-level" changes which can impact system stability later.
- OSRQ4: There are exceptions to the overall trend. In some cases, substantial level changes occur also across consecutive *Minor* versions and in *Pre*-releases (even though one might assume that architectures are more stable closer to a major release).

4.7 External replication

According to Carver's guidelines for replication studies [83], we need to report information about replication such as the level of interactions we had with original researchers, changes made to the original study and research questions we studied in the replication. We discuss these issues below.

4.7.1 Interaction with original researchers

While conducting the replication, we communicated with the authors of the original study via e-mail to clarify issues related to the use of ARCACE as well as the study design and details which were not reported in the original paper or other publicly accessible resources (e.g., the latest implementation of ARCADE, a technical manual, a detailed list of versions for each software system, the cluster files used in the original study). Therefore, rather than only reading the original paper we interacted with the original researchers closely.

4.7.2 Changes to the original study

Our subject systems were the same 14 open source projects (see Table 10) as in the original study. However, we studied more versions than the original study, i.e., a total of 810 versions published between 02/2000 and 11/2017. In the original study, there were 572 versions between 01/2001 and 06/2014. Furthermore, we changed the source code of ARCADE to generate additional Excel files with outputs. Two excel files are generated for each subject system (one file to display *a2a* values and the other for *cvg* values) containing all the possible pairs of versions with information such as the relevant version pair type and *cvg* and *a2a* values.

The original study used three recovery techniques to ensure construct validity. Among these techniques, ACDC and ARC both showed a higher accuracy compared to the other recovery techniques [11]. We applied only ACDC due to its simplicity and practical applicability. ACDC only requires .jar files as inputs, while other recovery techniques, e.g., ARC, requires shared common topic models. Considering a practitioners' point of view and since most developers have .jar files (source code may not be available for all systems), we use the technique that relies on jar file rather than source code and topic models.

Table 10 summarizes the systems we analysed, including the number of versions for each system and the timespan between the earliest and latest version in the original study and the replication. Note that most releases of JSPWiki had the same date in the online repository[1].

Table 10: Analysed systems

System	No. of Versions		Time Span	
	Original	**Replication**	**Original**	**Replication**
ActiveMQ	20	43	08/04 – 01/07	04/07- 10/17
Cassandra	127	159	09/09 – 09/13	02/10- 10/17
Chukwa	7	18	05/09 – 02/14	01/15- 07/16
Hadoop	63	106	04/06 – 08/13	04/06-11/17
Ivy	20	14	12/07 – 02/14	12/07- 12/14
JackRabbit	97	137	08/04 – 02/14	05/06-12/17
Jena	7	12	06/12 – 09/13	06/12-03/15
JSPWiki	54	35	10/07 – 03/14	07/13
Log4j	41	53	01/01 - 06/14	01/01 -11/17
Lucene	21	68	12/10 – 01/14	02/06 -10/17
Mina	40	42	11/06 – 11/12	11/06 -10/17
PDFBox	17	35	02/08 – 03/14	02/10-11/17
Struts	36	45	10/06 – 02/14	05/05-11/17
Xerces	22	43	03/03 – 11/09	02/00-10/14
Total	572	810	01/01 – 06/14	02/00–11/17

We included releases until 2017, except for a few systems:

- For Chukwa, the latest release available online was released in 2016.
- For Ivy, version 2.4.0-rc1 was the latest available version (included in our study) before releasing 2.5.0-rc1 recently in 2018. Since our study had finished by then, we did not include that version in our study.
- For Jena, we could not recover the architecture for later versions (3.0.0, 3.0.1, 3.1.1 etc.) because of an exception thrown from the implementation of ARCADE for creating cluster files using ACDC.
- Many of the versions of JSPWiki are dated with the same date and .jar files are not available from 2.9.0 to 2.10.2 in the source code of online repositories[2].
- The latest version of Xerces before 2.12.0 (released in April 2018 and therefore not included in our study) was version 2.11.0 (released in 2014)[3] and this version is included in this study.

[1] https://github.com/apache/jspwiki/releases
[2] https://github.com/apache/jspwiki/releases
[3] https://github.com/apache/xerces2-j/releases

4.7.3 Research questions of replication

Our replication addresses RQ2 of this thesis: ***How do software architectures change during maintenance and evolution?*** We investigate this RQ by decomposing it to two sub-questions (i.e., RQ2.1 and RQ2.2 of this thesis) in a way that those two sub-questions map the first two research questions of the original (i.e., replicated) study:

RQ2.1: To what extent do architectures change at system-level*?*
This RQ of the thesis replicates the first research question of the original study i.e., OSRQ1 (see Section 4.4.1).

RQ2.2: To what extent do architectures change at component-level?
This RQ of the thesis replicated the second research question of the original study i.e., OSRQ2 (see Section 4.4.1).

Answering above two RQs supports understand the other two research questions in the original study (OSRQ3 and OSRQ4): OSRQ3 of the original study is about comparing architectural changes at system and component level and OSRQ4 is about significant architectural changes in consecutive minor versions (which is also generally discussed in this replication). This means, answers for OSRQ3 and OSRQ4 are derived from first two research questions. Therefore, answering first two research questions provides sufficient knowledge to answer the other two questions and to understand whether or not findings related to OSRQ3 and OSRQ4 should be confirmed or refuted.

4.7.4 Replication procedure

Figure 20 depicts the work flow of the replication study.

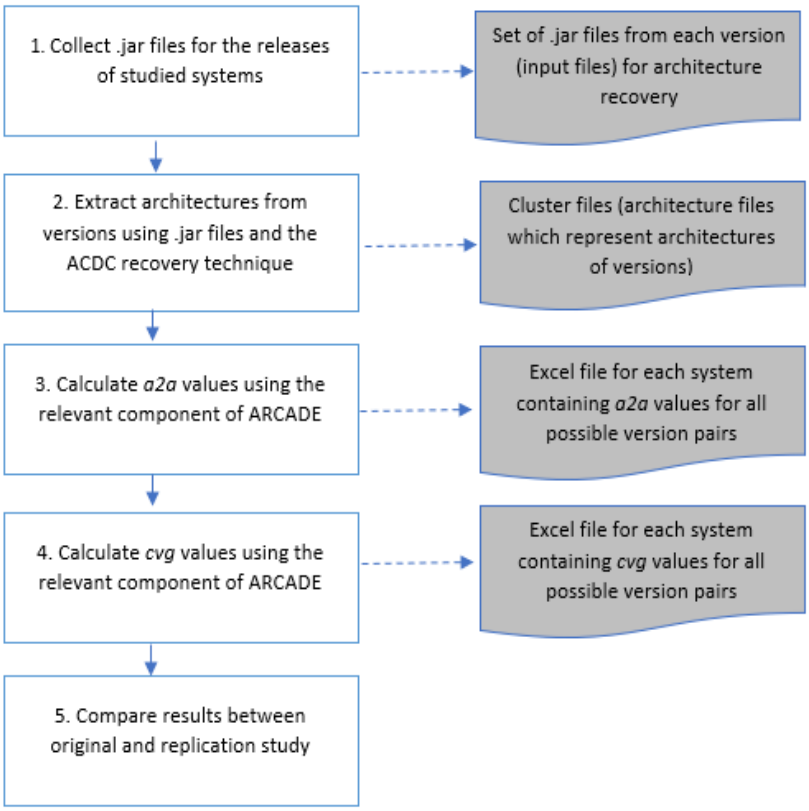


Figure 20: Replication procedure

4.8 Comparison of results

In this section, we report the results of our study and RQ2.1 and RQ2.2 of this thesis. Also, we compared our results with the results of the original study in two ways: First, we performed a full comparison (see Section 4.8.1.1) that compares results including all versions of the studied systems of the two studies (the versions analysed in the full comparison for each system are available in our online repository[4]; we also included the versions studied in the original study based on the version numbers of recovered cluster files available in the dataset received from authors of original study). Second, we performed an exact comparison (see Section 4.8.1.2) that compares results by considering only the versions of studied systems common to both studies.

[4]https://github.com/InfoResearch/Architecture-Decay-An-External-Replication/blob/master/List_of_analysed_system_versions.xlsx

4.8.1 RQ2.1: Architectural changes at system-level

Figure 21 shows the flow of analyses conducted to answer RQ2.1 and to compare our finding to the results of the original study. In Figure 21 "O" refers to the original study and "R" refers to the replication.

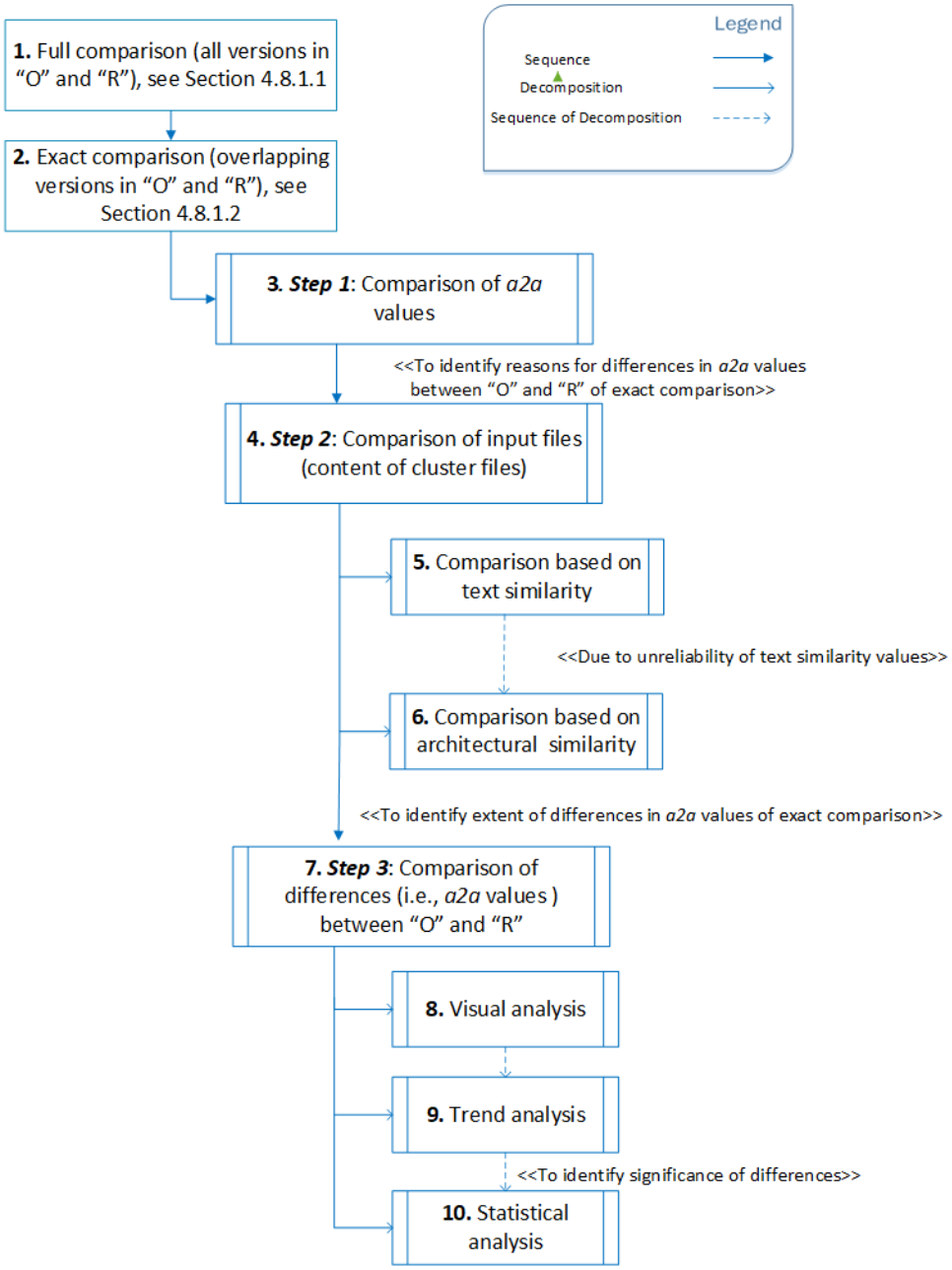


Figure 21: Analysis conducted to answer RQ2.1

4.8.1.1 Full comparison

We performed a full comparison by comparing the *a2a* values considering all the versions in our study with the respective results of the original study. Table 11 shows the average *a2a* value for all version pairs of a system in the replication (e.g., 67 for *MinMaj* in ActiveMQ indicates that the average *a2a* value for all *MinMaj* version pairs in ActiveMQ was 67%). The difference in average *a2a* values between the replication and the original study is shown in brackets. Empty cells indicate values that cannot be determined because the types of version pairs do not exist for a system in either the replication or in both studies. Also, if a particular type of version pair did not occur in the original study, differences are not included in the table (e.g., the original study did not include *Major* version pairs for PDFBox).

Table 11: Average *a2a* values (as percentage)

System	Average *a2a*				
	Major	**MinMaj**	**Minor**	**Patch**	**Pre**
ActiveMQ	-	67(-2)	94(-1)	100(0)	-
Cassandra	44(2)	80(0)	78(1)	99(0)	99(0)
Chukwa	-	-	76(-2)	-	99(4)
Hadoop	18(1)	69(-4)	91(5)	99(1)	98
Ivy	-	-	93(2)	-	98(-1)
JackRabbit	38(0)	80(4)	96(12)	100(9)	98(0)
Jena	-	-	92(4)	98(-1)	-
JSPWiki	19(1)	-	98(12)	100(2)	100(1)
Log4j	7(-2)	18(5)	95(31)	98(1)	88(3)
Lucene	50(38)	81(73)	95(-1)	100(2)	93(-1)
Mina	4(-24)	17(-13)	92(0)	100(1)	99(11)
PDFBox	41	54	96(-1)	99(2)	99
Struts	14	34	91(1)	100(1)	96
Xerces	59(38)	100(46)	88(-4)	95(12)	96
AVG	**32**	**60**	**91**	**99**	**96**

In the original study the overall trend of system-level architecture changes has been identified as $a2a_{Patch} > a2a_{Pre} > a2a_{Minor} > a2a_{MinMaj} > a2a_{Major}$ across the evolution paths of systems. As shown in Table 11, the results of the replication confirmed this trend. The last row ("AVG") shows the average of average *a2a* values (increasing from *Major* to *Patch*). Also, in more detail, rows for Hadoop, JackRabbit, Mina, PDFBox and Struts show the same trend. ActiveMQ, Chukwa, Ivy, Jena and JSPWiki also follow the same trend, but there are some empty cells. Other systems show exceptions to the overall trend which we discuss later in this section.

The overall trend indicates that substantial architectural changes occur across *Major* versions. Furthermore, *a2a* values for *MinMaj* pairs also indicate significant change. This could be since these pairs indicate that developers decided to move to the next major version. This move usually involves fewer changes than in *Major* versions. This explains $a2a_{MinMaj} > a2a_{Major}$ in the overall trend. The *Patch* versions usually involve bug fixes and local changes. *Minor* versions are usually subject to adding new features which require more changes than bug fixing. *Pre*-release versions are released to get feedback of a newly implemented feature before releasing an official minor or major version which involve less changes than between minor versions and therefore explains $a2a_{Patch} > a2a_{Pre} > a2a_{Minor}$.

However, there are some exceptions to the overall trend of architectural changes. For example, systems like Cassandra and Xerces show higher $a2a_{MinMaj}$ than $a2a_{Minor}$ values which is differed from the overall trend. This indicates that transitions between consecutive minor versions could also involve significant architectural changes. Another variance to the overall trend we observed is that there can be considerable architectural changes between *Pre*-release versions which are released to get feedbacks prior to an official release. *Pre*-releases may include system-level changes comparable to the changes between *Minor* versions. For example, for Lucene we noticed $a2a_{Pre} < a2a_{Minor}$ which shows that considerable architectural changes occur even at the last moment before a new official release. This exceptions to the common trend shown in our study related to Cassandra and Lucene were also observed in the original study.

The differences presented in Table 11 (values in brackets) highlight to what extent our results match or differ from the results of original study. Cassandra, Chukwa, Hadoop, Ivy, Jena PDFBox and Struts show only slight differences (only a difference of 0 to 5) since almost similar sets of versions of these systems have been analysed in both studies. However, there are systems which show larger differences:

- JackRabbit and Log4j show a larger difference (with differences of 12 and 31) for *Minor* version pairs because our data set includes ten more minor versions which are the latest minor releases of the evolution paths not analysed in the original study.
- JSPWiki, Lucene, Mina, and Xerces show considerably larger differences between the original study and our study for *Major* and *MinMaj* version pairs. The main reasons for these differences are: 1) Different interpretations of version pairs. In both the replication and the original study, the nearest

available patch or minor version to the major version is considered as major version in the absence of major versions. However, this nearest available patch or minor version is not the same version in the original study and the replication. For example, in Lucene, version pair 3.2.0 and 4.0.0 are considered a *Major* version pair in the original study (since 3.0.0 is not included in the original study and 3.2.0 is closest to version 3.0.0), but in our study the pair 3.0.0-4.0.0 is considered as *Major* since we included version 3.0.0. Therefore, for the same version pair (in this case the version pair should be 3.0.0-4.0.0 according to the definition of version pairs in Section 4.5), different versions were analysed in each study. 2) Different sets of versions have been analysed in each study. For example, for ActiveMQ, a completely different set of releases was analysed in our study compared to the original study. This was because older versions (the versions used in original study) were not available anymore at the time of replication. The original study analysed releases of ActiveMQ from versions 1.0.0 to 4.0.0 and our replication analysed versions from 4.1.1 to 5.15.2.

To summarize the full comparison, our replication confirmed the overall trends of architectural changes across the evolution path identified in the original study. However, there were many differences between average *a2a* values in the original study and the replication. Systems such as Cassandra, Chukwa, Hadoop, Ivy, Jena, PDFBox and Struts show only minor differences in *a2a* values. This is because the original study and the replication analysed almost identical sets of versions of these systems. However, there are systems which show bigger differences. The main reason for these differences could be the different sets of versions used in the two studies. Therefore, in next section we analysed the versions that appear in both the original study and the replication to see if the different sets of versions cause the differences of *a2a* values in the full comparison.

4.8.1.2 Exact comparison

We selected the subset of versions that appeared in both the original study and the replication. Such a subset of versions was not found for ActiveMQ since only later versions of ActiveMQ[5] were available for the replication and these versions were not used in original study. Therefore, we were unable to perform an exact comparison for ActiveMQ. Appendix C contains the list of all the versions of this comparison for each system with their percentage of overlapping versions between the original study and the replication.

[5] http://activemq.apache.org/download-archives.html (last access: May 2018)

First, we filtered the cluster files of common versions (see Appendix C) and grouped them into two sets: one set of cluster files from original study (we obtained these cluster files from the authors of the original study) and one set of cluster files from the replication (we generated the cluster files by us using the relevant component of ARCADE, i.e., *Recovery techniques*, see Figure 19). Then we performed a deeper analysis by following three steps:

- First, we compared *a2a* values of all version pairs using the same sets of version pairs from each study (i.e., Step 1: Comparison of *a2a* values, see Figure 21).
- In step 2 (Comparison of content of cluster files, see Figure 21), we compared the content of cluster files in the original study and the replication for the same version. This was to identify potential reasons for differences in *a2a* values.
- In the last step (Step 3: Comparison of differences, see Figure 21), we analysed the extent of differences in architectural changes identified in the replication compared to the original study.

We discuss these three steps in detail below.

In **step 1 (Comparison of *a2a* values),** we compared *a2a* values of version pairs in our study with those in the original study. Table 12 presents the results of the exact comparison which shows average *a2a* values and differences (values in brackets) between average *a2a* values for the same version pair used in the original study.

Table 12: Average *a2a* values for exact comparison

System	Average *a2a*				
	Major	**MinMaj**	**Minor**	**Patch**	**Pre**
ActiveMQ	Exact comparison was not performed				
Cassandra	65(0)	85(0)	74(0)	99(0)	98(0)
Chukwa	-	-	64(-14)	-	97(2)
Hadoop	18(1)	77(4)	92(5)	99(0)	-
Ivy	-	-	93(0)	-	-
JackRabbit	38(0)	80(0)	92(0)	100(0)	99(1)
Jena	-	-	91(3)	98(-1)	-
JSPWiki	-	-	100(5)	100(0)	92(-8)
Log4j	7(-2)	17(4)	64(26)	97(3)	85(-3)
Lucene	48(34)	58(44)	97(12)	100(0)	89(-4)
Mina	17(-11)	17(-13)	92(0)	99(0)	89(2)
PDFBox	-	-	97(0)	99(0)	-
Struts2	-	-	99(0)	99(0)	-
Xerces	59(7)	100(46)	92(-2)	100(2)	-
AVG	**36**	**62**	**88**	**99**	**93**

In **step 2 (Comparison of content of cluster files**), we compared the content of cluster files of the original study and the replication in two ways:

- **Text similarity:** We compared texts inside the cluster files using a text similarity measure *cosine distance*[6]. The purpose of this comparison was to find any differences between the content of cluster files recovered in our study and original study. Table 13 shows a comparison of descriptive statistics of text similarity of cluster files. Small distance values represent a high similarity between cluster files. To determine the percentage of similar cluster files (last column of Table 13), we used a threshold of 0.05 (i.e., clusters with a similarity value of +/-0.05 or less were considered similar).

Table 13: Comparison of cluster files based on text similarity

System	Mean	Median	Std. Deviation	Minimum	Maximum	% of similar cluster files (threshold 0.05)
ActiveMQ	Exact comparison was not performed					
Cassandra	.0006	0E-15	.006	-2E-16	.0598	99
Chukwa	.5217	.5161	.0855	.4038	.6712	0
Hadoop	.0032	.0009	.0064	-2E-16	.0287	100
Ivy	0E-15	0E-15	0E-15	-2E-16	1E-16	100
JackRabbit	.00004	0E-15	.0002	-2E-16	.0007	100
Jena	.0704	.0786	.0135	.0504	.0851	0
JSPWiki	.6042	.6207	.0852	.1726	.6614	0
Log4j	.0139	.0054	.0263	0E-16	.1168	88
Lucene	.1752	.2363	.1096	-2E-16	.2593	27
Mina	.1407	0E-15	.1977	-2E-16	.4198	66
PDFBox	5.65E-8	0E-15	1.33E-7	-2E-16	3.92E-7	100
Struts2	.0211	.025	.0052	.0149	.0266	100
Xerces	.0971	.1055	.0365	.0420	.1908	72

We set the threshold to 0.05 since it is close to zero and allowed us to ignore minor differences between clusters. Also, distance values close to zero may be due to rounding issues of the Java library used to compute the cosine distance. Therefore, clusters were considered similar, if cosine distance between two cluster files was within the range of -0.05 and 0.05.The average cosine distance values showed some differences (see column "Mean" in Table 13). This indicates that the cluster files in the original study and the

[6]https://commons.apache.org/proper/commons-text/apidocs/org/apache/commons/text/similarity/CosineDistance.html

replication are different for the same version of a software system. Furthermore, the results shown in Table 13 reflect the same differences as in the previous Section 4.8.1.2. According to the cosine distance, Cassandra, JackRabbit, PDFBox, and Ivy showed values smaller than 0.05 and also showed no differences in average *a2a* values between the two studies (see difference values in Table 12).

Based on this comparison we found that some cluster files for the same version were considerably different (e.g., JSPWiki, Lucene, Chukwa, Mina, etc. as shown in Table 13). However, we found that some differences identified through text similarity measure were due to the order of the lines of text inside the cluster files (see Section 4.4.4), i.e., the order of text strings impacted the similarity metrics. For example, in Hadoop 0.4.0, cluster files between the two studies showed a difference based on text similarity. When examining the text inside the cluster files, we found that the text inside the cluster files for the same versions in the two studies looked different in their "default view" generated by the ARCADE tool. However, both files had the same number of lines and when sorting lines of text in alphabetical order, we found that both files (cluster files generated for Hadoop 0.4.0 in original study and replication) have the same lines with the same sets of words. Therefore, the difference based on text similarity was due to the different order of its lines or different spacing between words. In a next step we compared cluster files based on architectural similarity. The cluster files for the same version of a system used in the original study and the replication should show no architectural changes (even if the order of strings differs).

- **Architectural similarity:** We computed *a2a* values between cluster files i.e., for a version in the original study and the replication (note that this is different from the original purpose of the *a2a* metric to compute architectural changes between two different versions). This comparison was to ensure that similar architectures have been recovered for same release in both studies. Table 14 shows a comparison between descriptive statistics of architectural similarity between cluster files with the percentage of similar clusters for each system. If *a2a* value between two clusters was above the threshold value 90, the clusters were considered similar. We set 90 as threshold because it helps us to ignore minor differences between cluster files, e.g., due to rounding errors.

Table 14: Architecture similarity of cluster files

System	Mean	Median	Std. Deviation	Minimum	Maximum	% of similar cluster files
ActiveMQ	Exact comparison was not performed					
Cassandra	99.38	100	5.43	46.3	100	99
Chukwa	15.32	15.64	0.83	13.73	16.08	0
Hadoop	93.15	95.3	6.43	70.5	100	78
Ivy	100	100	0.00	100	100	100
JackRabbit	99.64	100	1.36	94.5	100	100
Jena	73.35	74.15	2.22	69.3	75	0
JSPWiki	12.78	13.55	2.36	7.9	15.8	0
Log4j	78.18	72.7	17.64	28.8	100	41
Lucene	39.03	17.10	38.06	15.7	100	27
Mina	96.33	100	5.22	88.6	100	67
PDFBox	99.87	100	0.51	97.9	100	100
Struts2	62.01	49.7	14.66	48.5	77.5	0
Xerces	48.09	45.65	11.89	26.0	68.8	0

The results showed mismatches between *a2a* values for some systems, even though we compared the architectures of cluster files for the same version. Cassandra, JackRabbit, PDFBox and Ivy showed no differences in cluster files (*a2a* similarity values are between 99 to 100) which is consistent with the results of previous textual comparison (lower average cosine distance values, i.e., values lower than 0.05 in Table 13) as well as the average *a2a* values comparison (which showed no differences in Table 12). Chukwa, JSPWiki, Lucene and Xerces showed considerably lower similarities compared to other systems.

Based on the two types of comparison (i.e., the comparison based on text and architectural similarity) performed in this step to compare content of cluster files, we found that there were differences in the input files to compute architectural change between versions of systems. These differences appear even though we analysed input files for the same version. The reason for these differences is that the recovery of architectures for a system is based on manually selected .jar files from the software projects' repositories. There is a possibility that different .jar files have been selected to generate cluster files for the same version from each study. For example, when generating cluster files for systems like Cassandra we contacted the authors of the original study. Since Cassandra has many .jar files (around 30 files in each release), it was uncertain which files to select. Based on the advice of the original authors, we selected only the .jar files with the prefix "cassandra" in the

name of the file. Therefore, the average *a2a* values for Cassandra were same between original study and the replication.

Since there were differences of architectural changes between the original study and the replication (and we found reasons for these differences), our next focus was to analyse the differences between the two studies to see whether the overall architecture changes follow at least a consistent trend in both studies.

In **step 3 (Comparison of differences),** we analysed the nature and extent of the differences in architectural change identified in the original study and the replication. Here, we analysed trends of architectural changes because even though the selection of .jar files affect concrete *a2a* values, it may not always change the trend of the architecture evolution of systems (as long as .jar files for the same versions are compared). Therefore, this analysis was performed to see if at least the trend remains same (even though the actual *a2a* values differ), especially for the systems with different recovered architectures even for the same release. We analysed the trend in two ways: visual comparison and comparison of percentage of similar trends. Then, a statistical analysis was performed to analyse statistical significance of differences in *a2a* values between the two studies.

1. **Percentage of similar trends:** We calculated the percentage of similar trends between the original study and the replication for each system. To determine trends, we checked whether *a2a* values increased, decreased or remained constant between sequential version pairs and whether the same trend appears in the original and the replicated study.

   The results presented in Table 15 show the number of total trends (i.e., the total number of increasing, decreasing and constant trends across the release sequence [i.e., the number of trends is the number of version pairs minus 1]), the number of similar trends, and the percentage of similar trends between the original study and the replication. Across the release sequences of all systems, every system showed 70%-100% similarity in trends except for Chukwa, JSPWiki and Xerces (see the last column of Table 15).

Table 15: Similarity of trends (*a2a*) in original study and replication

System	Total No. of trends	No. of similar trends	% of similar trends
ActiveMQ	Exact comparison was not performed.		
Cassandra	96	93	96.88
Chukwa	4	2	50.00
Jena	8	6	75.00
JSPWiki	26	15	57.69
Hadoop	62	50	80.65
Ivy	2	2	100.00
JackRabbit	75	75	100.00
Log4J	37	35	94.59
Lucene	13	11	84.62
Mina	36	33	91.67
PDFBox	15	14	93.33
Struts2	11	9	81.82
Xerces	16	10	62.50

2. **Visual comparison:** A visual comparison of patterns of architectural changes was performed by plotting the *a2a* values against system versions for each system (see Appendix D). As an example, see the graph for PDFBox in Figure 22. Even though some of the *a2a* values in our study were different from the original study, most version pairs followed similar trends. Exceptions were JSPWiki and Xerces. The graph for Xerces (Xerces showed the biggest difference of *a2a* values in the exact comparison) is shown in Figure 23. ("O" refers to original study and "R" refers to replication).

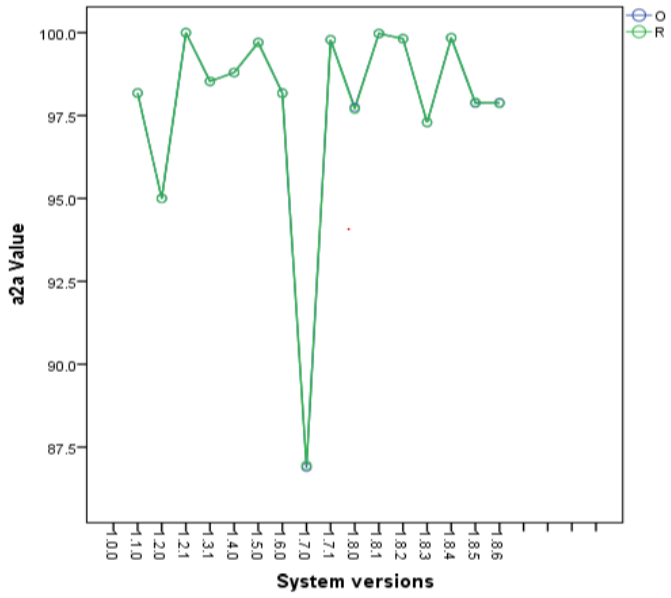


Figure 22: *a2a* values for PDFBox

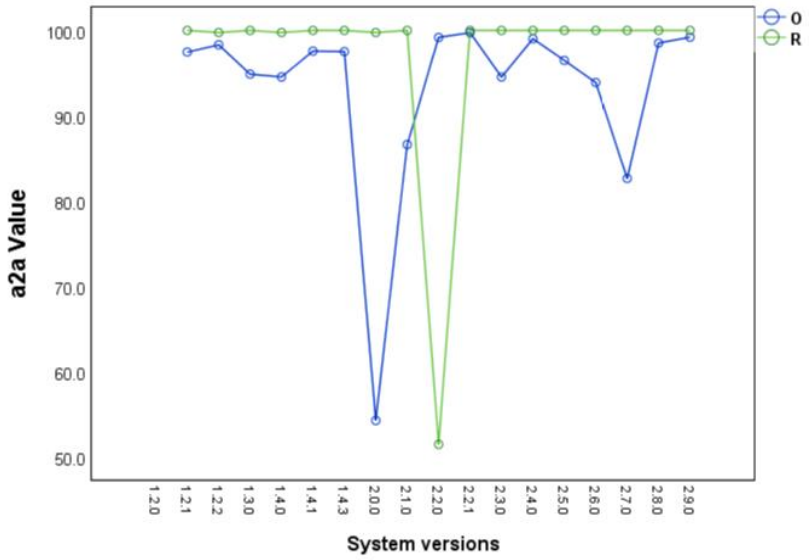


Figure 23:*a2a* values for Xerces

3. **Statistical comparison:** We then analysed the differences of *a2a* values between original and replication statistically. We first performed a normality test (*Shapiro-Wilk*) to check the distribution of the *a2a* values of sequential version pairs in the sequence of releases of all the 14 systems in both the original study and the replication. Since the data were not normally distributed (there was enough evidence to reject the normality hypothesis at the level of α=0.05) and for some systems the number of *a2a* values was too small to determine their distribution, two different statistical tests were applied to test statistical significance of the differences.

   First, the *two-sample t-test* was applied for the systems with a big enough sample size (Cassandra, JSPWiki, Hadoop, JackRabbit, Log4j, Mina, PDFBox and Xerces which have more than 15 common version pairs). The results showed a statistically significant difference only for Hadoop. Since Hadoop showed a statistically significant difference, the effectiveness of the difference was analysed by calculating effect size using *Cohen's d distance*. The effect size for Hadoop was 0.322 (see Table 16) which is in the middle of the benchmarks introduced by *Cohen* (small effects [0.2] and medium effects [0.5]) indicating that the difference in Hadoop is trivial.

   Second, a non-parametric test (*Wilcoxon test*) was applied to all the systems again in case if there were outliers in the data sets (a non-parametric test would perform better than a parametric test when outliers). However, the results for *Wilcoxon test* gave *p*-values larger than .05 for all

the systems including Hadoop (though Hadoop showed a statistically significant difference according to parametric test) indicating that there is no statistically significant difference. Even though there was a statistically significant difference between the original study and the replication for Hadoop (according to parametric test) the effect size is small (according to *Cohen's d* distance). Therefore, we conclude that there is no statistically significant difference between the *a2a* values obtained in the original study and the replication.

Table 16: Statistical tests of differences in *a2a* values in original study and replication

System	Statistical Test		
	Paired sample t-test	**Wilcoxon test**	**Cohen's d distance**
ActiveMQ	Exact comparison was not performed.		
Cassandra	0.366	0.717	-
Chukwa	Small sample size	0.705	-
Jena	Small sample size	0.086	-
JSPWiki	0.794	0.078	0.0508
Hadoop	0.013	0.613	0.3222
Ivy	Small sample size	1	-
JackRabbit	0.179	0.809	0.1569
Log4J	0.481	0.074	0.1156
Lucene	Small sample size	0.552	-
Mina	0.276	0.281	0.1819
PDFBox	0.971	0.715	0.0091
Struts2	Small sample size	0.875	-
Xerces	0.729	0.381	0.0854

To summarize the exact comparison, the exact comparison compares the same sets of versions from original study and the replication. The results of the exact comparison also showed some differences between the original study and the replication. These differences were fewer than the differences which occurred in the full comparison. However, there were differences between the original study and the replication, so we explored reasons for these differences.

An analysis of the cluster files (i.e., textual and architectural comparison) revealed that there are differences in the content of cluster files. These differences were consistent with each other in both architectural and textual comparison of clusters. We found the reason for the differences in *a2a* values between the two studies is due to the differences in input files (cluster files). We further found that the reason for these differences in input files is due to different .jar files that have been selected to generate cluster files. Since the required .jar files have been selected manually in both the original and the replicated studies, there is a possibility that different .jar files have been selected to generate cluster files for the same version of both studies.

While there are differences between average *a2a* values, we confirm that our study gave the same results as in original study for systems Cassandra, JackRabbit, Ivy and PDFBox since these systems showed no differences in comparison of average *a2a*, text and architectural similarity of cluster files. However, even though there were differences in *a2a* values for other systems (which were not statistically significant), the same trend of architecture evolution was confirmed in both studies. Also, the patterns of architectural changes across the sequential version pairs in both studies were same for majority of the version pairs (see Table 15).

4.8.1.3 Summary of RQ2.1

To answer RQ2.1 of this thesis which investigates architectural changes at system level, we compared our results with the results of the original study and performed two types of comparison: a full comparison (see Section 4.8.1.1) and an exact comparison (see Section 4.8.1.2).

In the full comparison, where we compared average *a2a* values considering all versions in our study with the respective results of the original study, we confirmed the overall *trend* of architectural changes across the evolution path identified in the original study. We could not confirm the actual average *a2a* values in the original study and found differences in average values between the two studies. The reason for these differences were the different sets of versions in the original study and the replication. Therefore, we also performed an exact comparison considering only the versions that appear in both the original study and replication. In the exact comparison, we found fewer differences in average *a2a* values than in full comparison and we could confirm *a2a* values in the original study for some systems. Based on these results of the exact comparison, we conclude that a) Cassandra, JackRabbit, Ivy and PDFBox showed the same *a2a* values in the original study and the replication, b) other systems showed differences due to the differences in input files (architectural files), but still follow the same overall trend of architecture evolution as in the original study, and c) the differences in *a2a* values between the two studies are not statistically significant.

4.8.2 RQ2.2: Architectural changes at component-level

To investigate architectural changes at component-level, we computed the average *cvg* values for each version pair (*s*, *t*) with *s* as the source version and *t* the target version. *cvg* values indicate the percentage of existing components in a cluster (architecture of a version) after evolving from previous (*s*) to current version (*t*).

Also, we calculated *cvg* for the inversion (*t*, *s*) of a pair. This allows us to determine the percentage of components newly added after evolving from the source version (*s*) to target version (*t*). We followed the same two types of comparisons as for RQ2.1: a full comparison and then an exact comparison.

4.8.2.1 Full comparison

Table 17 shows the average *cvg* values for the full comparison in the replication with differences to the original study in brackets. Differences are not mentioned when a particular type of version is not available in the original study. As also explained in the original study, we noticed that there are dissimilarities between some version pairs and their corresponding inverse pairs. This means, the percentage of components remaining and added when evolving to the next version do not complement each other. This is due to the system's increase in size during the evolution. For example, for ActiveMQ, cvg_{MinMaj} (*s*, *t*) is 38% and cvg_{MinMaj} (*t*, *s*) is 29%. This indicates that after iterating to the next major version, 38% of components of the previous version (in this case, the immediately preceding minor version) remained and 71% of the components of the next version (i.e., 100 - 29) were added in that next version (in this case, the newly introduced major version). This means, the total number of components added and remaining (38% + 71%) is more than 100%, because ActiveMQ grew by an average of 31% (cvg_{MinMaj} (*s*, *t*)/ cvg_{MinMaj} (*t*, *s*)) in the number of components during the evolution to the next major version. Overall, the difference between average *cvg* values for version pairs and their inverse pairs in all subject systems (see last row in Table 17) is between 0% (*Patch* versions) and 7% (*Major*). This range is nearly equal to the range mentioned in the original study. 0% for *Patch* versions (which was same value in the original study for *Patch* versions) shows that patch versions do not grow considerably.

Overall, both the original study and the replication show extensive component-level changes for *Major* and *MinMaj* version pairs and considerable stability for *Minor*, *Patch*, and *Pre*-releases.

According to the findings of the original study, *cvg* values for each version pair and its inverse pair should follow the same overall trend*:* $cvg_{Patch} > cvg_{Pre} > cvg_{Minor} > cvg_{MinMaj} > cvg_{Major.}$ As shown in Table 17, the results of the replication also show that trend, except for Log4j and Xerces. However, individual *cvg* values differ in the original study and the replication. These differences were already

expected since different sets of versions have been analysed in the original study and the replication.

Since there were considerable differences with *cvg* values in some cases, we compared the *cvg* values using the same sets of versions from the original study and the replication as explained for the exact comparison for RQ2.1 (see Section 4.8.1.2). In next section we discuss the results of the exact comparison of *cvg* values.

Table 17: Average *cvg* values

System	Major		MinMaj		Minor		Patch		Pre	
	(s, t)	(t, s)	(s, t)	(t, s)	(s, t)	(t, s)	(s, t)	(t, s)	(s, t)	(t, s)
ActiveMQ	-	-	38(-23)	29(-19)	90(-5)	89(-3)	100(0)	100(0)	-	-
Cassandra	11(6)	8(4)	61(2)	54(1)	53(1)	45(-1)	99(1)	99(0)	97(-1)	97(-1)
Chukwa	-	-	-	-	63(0)	87(33)	-	-	99(13)	100(14)
Hadoop	0(0)	0(0)	45(-9)	43(-3)	88(5)	81(7)	98(3)	98(0)	96	96
Ivy	-	-	-	-	93(26)	87(30)	-	-	95(-5)	95(-1)
JackRabbit	16(0)	7(0)	53(0)	57(0)	94(7)	91(10)	99(1)	99(2)	98(2)	97(1)
Jena	-	-	-	-	91(10)	86(12)	98(2)	95(-1)	-	-
JSPWiki	26(26)	4(4)	-	-	99(61)	96(61)	99(14)	99(15)	96(-2)	87(-11)
Log4j	0(0)	0(0)	0(0)	0(0)	94(65)	91(70)	96(2)	95(2)	78(-7)	75(-7)
Lucene	9(9)	9(9)	69(69)	74(74)	89(2)	87(3)	99(1)	100(2)	86(-13)	72(-27)
Mina	0(4)	0(2)	0(-4)	0(-2)	78(0)	78(0)	99(0)	99(0)	98(11)	97(17)
PDFBox	14	11	32	29	92(-2)	91(-1)	99(4)	99(5)	100	99
Struts	0	0	21	13	74(-5)	70(-13)	99(3)	99(3)	91	87
Xerces	60(60)	27(27)	100(80)	100(84)	83(0)	76(-5)	93(7)	93(10)	95	88
AVG	**14**	**7**	**42**	**40**	**84**	**83**	**98**	**98**	**94**	**91**

4.8.2.2 Exact comparison

An exact comparison of *cvg* values was performed for RQ2.2 using same set of versions from the original study and the replication. The results of exact comparison are presented in Table 18.

There were still differences between the values obtained in the original study and the replication, but the amount of differences (see values inside brackets in Table 18) was smaller compared to the differences in the full comparison (see differences in Table 17). As discussed for RQ2.1, the reason for these differences were differences in input files (these differences in input files were already analysed in Section 4.8.1.2). As the next step we analysed these differences further to understand the extent of these differences using the same types of analyses as in Section 4.8.1.2.

Table 18: Average *cvg* values for exact comparison

System	Major		MinMaj		Minor		Patch		Pre	
	(s, t)	(t, s)	(s, t)	(t, s)	(s, t)	(t, s)	(s, t)	(t, s)	(s, t)	(t, s)
ActiveMQ	Exact comparison was not performed									
Cassandra	33(0)	29(0)	66(2)	57(0)	50(0)	45(1)	99(1)	99(0)	97(0)	98(1)
Chukwa	-	-	-	-	43(-20)	75(21)	-	-	96(11)	100(13)
Hadoop	0(0)	0(0)	63(9)	45(-1)	92(7)	82(6)	99(1)	98(0)	-	-
Ivy	-	-	-	-	95(0)	87(0)	-	-	-	-
JackRabbit	16(0)	7(0)	53(0)	57())	89(0)	83(0)	99(0)	99(0)	98(4)	98(3)
Jena	-	-	-	-	88(7)	82(8)	98(2)	96(-1)	-	-
JSPWiki	-	-	-	-	100 (0)	100(0)	99(0)	99(0)	94(-5)	82(-18)
Log4j	0(0)	0(0)	0(0)	0(0)	33(33)	23(23)	95(5)	94(5)	80(-3)	76(-4)
Lucene	31(31)	21(21)	47(47)	37(37)	85(1)	82(0)	100(0)	100(0)	79(-18)	59(-38)
Mina	0(-4)	0(-2)	0(-4)	0(-2)	78(0)	78(0)	99(0)	99(1)	78(-10)	68(-13)
PDFBox	-	-	-	-	94(0)	92(0)	98(0)	98(0)	-	-
Struts2	-	-	-	-	95(-5)	90(-10)	99(0)	98(-1)	-	-
Xerces	60(43)	28(17)	100(83)	100(89)	90(2)	81(-4)	100(6)	100(6)	-	-
AVG	**20**	**12**	**47**	**42**	**79**	**77**	**99**	**98**	**89**	**83**

1. **Percentage of similar trends:** Percentages of similar trends were calculated at component-level and the results are presented in Table 19. This is the same trend analysis performed for RQ2.1 (see Table 15) for *a2a* values except that the percentage was calculated for inverse version pairs as well.

Table 19: Similarity of trends (*cvg*) in original study and replication

System	Total No. of trends	No. of similar trends		% of similar trends	
		(s, t)	(t, s)	(s, t)	(t, s)
ActiveMQ	Exact comparison was not performed.				
Cassandra	96	95	92	98.96	95.83
Chukwa	4	2	3	50	75
Jena	8	6	7	75	87.5
JSPWiki	25	17	15	68	60
Hadoop	62	46	48	74.19	77.42
Ivy	2	2	2	100	100
JackRabbit	75	74	74	98.67	98.67
Log4J	37	27	28	72.97	75.68
Lucene	13	7	8	53.85	61.54
Mina	36	27	26	75	72.22
PDFBox	15	15	15	100	100
Struts2	11	10	9	90.91	81.82
Xerces	16	5	6	31.25	37.5

Across the release sequences of all systems, every system showed 70%-100% of similar trends except for Chukwa, JSPWiki, Lucene and Xerces (see the last column in Table 19). These systems also showed lower

similarity when analysing the content of cluster files between the original study and the replication (see Section 4.8.1.2). Chukwa, JSPWiki and Xerces also showed lower percentage of similar trends (which is less than 70%) than other systems when comparing *a2a* values (see Table 15).

2. **Visual comparison:** For the visual comparison, we plotted *cvg* values (see Appendix E) for the original study and the replication in two graphs for each system: one showing the pattern of changes in version pairs (*s*, *t*) and the other one showing the pattern in inverse pairs (*t*, *s*). For example, the following graphs (see Figure 24 and Figure 25) for Xerces (Xerces showed highest differences in the exact comparison) have been used for visual comparison. The relevant graphs seem shifted a little for most of the versions while following the same trend as in the original study. This shift can be observed in all systems except for Cassandra, Ivy, JackRabbit and PDFBox which were with no differences in average *cvg* values for most of the versions in exact comparison (see Table 18) and nearly 100% similar trends (see respective rows in Table 19) between the original study and the replication. The shift shown in the visual comparison is probably because of the differences in input (.jar) files that were analysed to recover architecture from source code of releases as already discussed for RQ2.1 in Section 4.8.1.2.

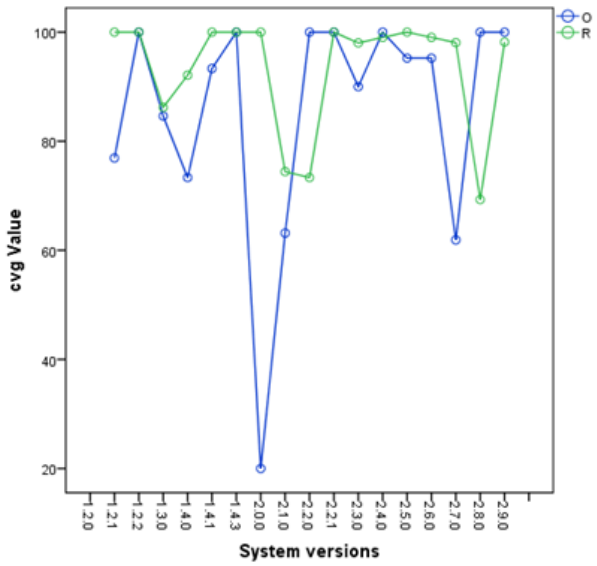


Figure 24: *cvg* (*s*, *t*) values for Xerces

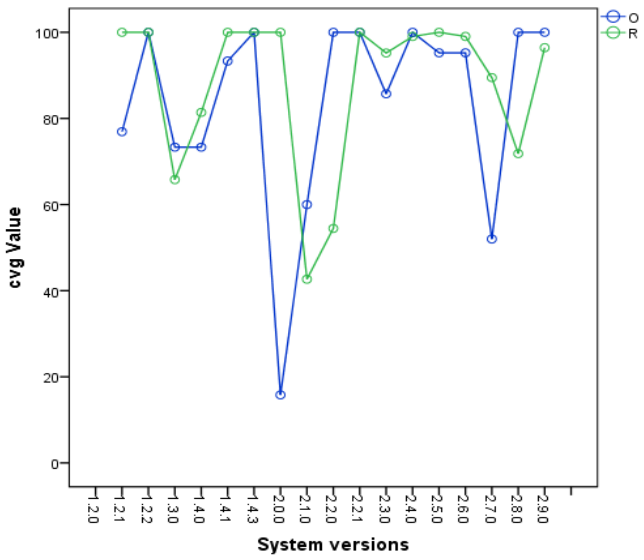


Figure 25: *cvg* (*t, s*) values for Xerces

3. **Statistical comparison:** Since there are differences in average *cvg* values as well as in trends of changes in *cvg* values between the two studies, we checked whether these differences are statistically significant by conducting the same statistical analysis performed for RQ2.1. For RQ2.2, we conducted two separate statistical analyses across all systems: one for version pairs (see Table 20) and the other for the inverse of version pairs (see Table 21).

   First, the normality test was performed to check the distribution of data. This test was applied on the two data sets (original and replication) for each system. All the systems except Chukwa and Ivy followed non-normal distributions (*p*-values smaller than α=0.05, see Table 20 and Table 21). Then, to analyse statistically significant differences, the *paired sample t-test* and the non-parametric test (*Wilcoxon* test) were applied. According to the t-test, Hadoop and JSPWiki showed a statistically significant difference (*p*-values smaller than α=0.05) for version pairs *s*, *t* (see Table 20). For PDFBox, it is impossible to calculate *p*-value as the standard error of the difference was zero (see differences for PDFBox in Table 18). All *p*-values for inverse pairs (*t*, *s*) for all systems were larger than 0.05 (see Table 21) indicating that the differences for *t*, *s* pairs are not statistically significant. Since differences were statistically significant for only two systems (Hadoop and JSPWiki) according to t-test, we calculated effect size (see below) to analyse the impact of these differences on findings of the original study.

The non-parametric test was applied for systems for which the t-test was not performed because the sample size was too small. Also, this test was performed for Hadoop and JSPWiki (i.e., the systems showed statistically significant differences for t-test) to check the results of parametric test in case the general assumptions of a t-test (e.g., insufficient sample size with non-normal distribution of data, outliers of data etc.) are violated. The results for *s, t* pairs showed a statistically significant difference for Jena in addition to Hadoop and JSPWiki since these three systems showed *p*-values of less than 0.05. The results for *t, s* version pairs showed no statistically significant differences since all *p*-value were higher than 0.05 (see Table 21).

The corresponding effect size was calculated for the systems which showed statistically significant difference in the *paired sample t-test* (i.e., Hadoop and JSPWiki) using *Cohen's d* and also for systems that showed a statistically significant difference in the *Wilcoxon test* (i.e., Jena, Hadoop and JSPWiki) using *Rosenthal's formula*. Hadoop and JSPWiki showed medium level effect sizes according to *Cohen's d* since the effect size was within the range of 0.2 and -0.5. According to *Rosenthal's formula*, Hadoop, JSPWiki and Jena showed small (-0.1 ≤ *p*-value), medium (-0.3 ≤ *p*-value) and large (-0.5 ≤ *p*-value) effect sizes, respectively. Effect sizes were not calculated for inverse pairs since the differences were not statistically significant.

Table 20: Statistical test results for *cvg* (*s, t*)

System	Statistical test					
	Normality test		Paired sample t-test		Non-parametric test (Wilcoxon test)	
	Replication	Original study	Paired sample t-test	Effect size(d)	Wilcoxon test	Effect size(r)
ActiveMQ	Exact comparison was not performed.					
Cassandra	0.000	0.000	0.321	-	-	-
Chukwa	0.040	0.066	-	-	0.276	-
Jena	0.000	0.001	-	-	0.008	-0.62
JSPWiki	0.000	0.000	0.044	0.416	0.013	-0.34
Hadoop	0.000	0.000	0.023	0.294	0.026	-0.01
Ivy	0.000	0.000	-	-	1.000	-
JackRabbit	0.000	0.000	0.169	-	-	-
Log4J	0.000	0.000	0.411	-	-	-
Lucene	0.000	0.000	-	-	0.866	-
Mina	0.000	0.000	0.389	-	-	-
PDFBox	0.001	0.001	-	-	1.000	-
Struts2	0.000	0.000	-	-	0.249	-
Xerces	0.000	0.000	0.200	-	-	-

Table 21: Statistical test results for *cvg* (*t, s*)

System	Normality test		Paired sample t-test	Non-parametric test (Wilcoxon test)
	Replication	Original study		
ActiveMQ	Exact comparison was not performed.			
Cassandra	0.000	0.000	0.752	-
Chukwa	0.304	0.000	-	0.157
Jena	0.000	0.029	-	0.086
JSPWiki	0.000	0.000	0.240	-
Hadoop	0.000	0.000	0.054	-
Ivy	0.559	0.559	-	1.000
JackRabbit	0.000	0.000	0.183	-
Log4J	0.000	0.000	0.604	-
Lucene	0.000	0.000	-	0.398
Mina	0.000	0.000	0.417	-
PDFBox	0.000	0.000	-	1.000
Struts2	0.000	0.000	-	0.866
Xerces	0.000	0.000	0.522	-

To summarize the results of statistical test, the results confirmed that differences in average *cvg* values between the original and our study are not statistically significant for inverse version pairs (*t, s*) of all the systems. For version pairs (*s, t*), there was no statistically significant difference for most of the systems except JSPWiki, Hadoop and Jena. Among these systems, the impact of differences in Hadoop is negligible since the effect size is small.

4.8.2.3 Summary of RQ2.2

In summary, there were fewer differences in the exact comparison than in the full comparison. While there were differences in *cvg* values for some systems, we confirm the average *cvg* values of most of the versions of Cassandra, Ivy, JackRabbit and PDFBox (see Table 18) where the differences were small (0 to 4). These differences were already expected due to the differences in input files identified when answering RQ2.1 (see Section 4.8.1.2). Since there were differences, the extent of these differences was analysed by comparing trends. The results showed 70%-100% of similar trends for all systems, except for Chukwa, JSPWiki, Lucene and Xerces. After performing statistical analyses, we found that differences in average *cvg* values between the original study and our replication are not statistically significant for inverse version pairs (*t, s*) of all the systems and for version pairs (*s, t*) of all systems except JSPWiki, Hadoop and Jena. However, the impact of differences in Hadoop is not significant enough to reject the basic finding of the original study since effect size is small. For JSPWiki and Jena, we

could not confirm the findings in the original study, but still we confirm the trend of architectural evolution in the original study by considering the visual comparison of graphs of the two systems (see Appendix E).

Based on the results of the exact comparison we conclude that a) Cassandra, JackRabbit, Ivy and PDFBox showed the same average *cvg* values in the original study and the replication, b) other systems showed differences due to the differences in input files (architectural files), but still follow the same overall trend of architecture evolution as in the original study, and c) the differences in *cvg* values between the two studies are not statistically significant for inverse version pairs (*t, s*), but in version pairs (*s, t*) there were statistically significant differences for JSPWiki, Hadoop (small effect size) and Jena.

4.9 Discussion of results

4.9.1 Comparison with original study

For architectural changes at both system and component level, we could not confirm individual *a2a* and *cvg* values in the original study. However, there were no statistical significant differences at system-level changes between the two studies except for Hadoop. Since the effect size was small, the difference in Hadoop is negligible.

For architectural changes at component-level, we found that there is no statistically significant difference in the *cvg* values for inverse version pairs (*t, s*) for all the systems. For version pairs (*s, t*), there were statistically significant differences in three systems: Hadoop showed a negligible difference (effect size is small) as at system-level and JSPWiki and Jena showed statistically significant differences between the two studies with medium and large effect sizes. However, these differences did not contradict the findings of the original study since the overall trend of architectural changes between the two studies is consistent. We confirm the basic findings of the original study, i.e., the overall trend of architecture evolution (*Patch* < *Pre* < *Mino*r < *MinMaj* < *Major*) across the release sequence of all systems, which is common to both system and component level.

Furthermore, we confirm that at both system-level and component-level, substantial architectural changes usually happen between *Major* and *MinMaj* version pairs. However, there are some exceptions to this: There can be larger architectural changes between minor versions in the same major version where $a2a_{Minor}$ is closer

to $a2a_{MinMaj}$ or even lower than $a2a_{MinMaj}$ and there may be considerable architectural changes among pre versions where $a2a_{Pre}$ is lower than $a2a_{Minor}$ (see Section 4.8.1.1). This indicates that the versioning schemes used by developers currently do not consider the architectural impact of the changes. Versioning schemes that consider the extent of architectural changes would help understand differences in the releases more effectively. For example, if the format of a version is x. y. z where x, y, z are non-negative integers, x could indicate the number of the major version, y could indicate the number of the minor and z could indicate the patch number (as in the current versioning scheme as explained in Section 4.5). When changes to a system include extensive changes in a system's architecture, the next version could increment the major version, while smaller architectural changes than in major releases could increment minor versions and lowest level changes in the system architecture could increment the patch version. This would help developers keep track of a system's architecture throughout the evolution path and stay vigilant on potential architecture decay.

In addition, in the full comparisons where we analysed versions including recent releases, we did not find any differences in the overall trend. This indicates that architectures do not become stable over time.

Note that above discussion on comparing our results with the original study is only based on the first two research questions of the original study (i.e., OSRQ1 and OSRQ2) as explained in Section 4.7.3. We focused only those two research questions in our replication, since answering to those two provides sufficient knowledge to understand and determine answers of remaining two research questions (OSRQ3 and OSRQ4) of the original study. In general, we can confirm the findings of OSRQ3 and OSRQ4 since we confirmed the findings of the first two RQs. OSRQ3 is about comparing system and component level changes i.e., whether each type of architectural change (system-level, component-level) occur concurrently. The main finding of OSRQ3 in the original study was that even though architectural changes at both system and component level followed the same overall trend, the extent of that change differs. Furthermore, these differences steadily grow from *Patch* and *Pre*-release versions (where the two metric values return almost similar values) to *Major* versions where *cvg* values are notably lower compared to *a2a* values. We confirm this finding in our study as well, see AVG rows in Table 11 and Table 17. For example, systems Hadoop, Log4j, Mina, and Struts showed 0% for *Major* version in *cvg* values but not for *a2a* values. The findings of OSRQ4 in the original study was that major architectural changes can occur in consecutive minor version pairs within a major version. This was already discussed in general

and confirmed when discussing about exceptions to the overall trend (see Section 4.8.1.1) and also when summarizing the results early in this section. Therefore, we confirm the results of the last two research questions of the original study even though we did not explicitly answer these in the replication.

4.9.2 Lessons learned about replications

While conducting the replication, we have identified several lessons learned that might be useful for other researchers:

Use of guidelines: It may be difficult to decide how many details of the original study should be included in the report of a replication, since many parts of the original study and the replication are the same. Therefore, as explained in the guidelines proposed by Carver [83], we have provided enough details of the original study to understand the work of the original study and more details about the adjustments in the replication. However, while reporting the replication we found some issues that guidelines do not address. For example, Carver's guidelines do not include a separate section for discussing the results of the replication. Before comparing results of studies, we believe that discussing the results of the replication separately could guide the reader through the replication and help them better understand the differences between an original study and a replication and in particular how studies were compared.

Interaction with authors of original study: Interaction with the authors of the original study was essential to fully understand the original study. As discussed earlier, we received technical and methodological guidance from the original authors. In general, it is necessary to have as much information about the original study as possible to ensure that the original and replication studies are comparable. Proper understanding of the approach and systems used in the original study and resulting deviation was also a challenge reported in a replication study on measuring architectural quality by Reimanis et al.[103].

Adjustments in the replication: Even though the original study and the replication look similar, there were considerable differences between the results of the two studies. However, we could trace these differences back to reasons (e.g., differences in metrics values are because of different .jar files that were analysed). Identifying reasons for differences was possible because relatively few adjustments were made to the replication. This was also one lesson learned from a replication conducted by Cecilia et al.[110].

Availability of experimental subjects: According to Sioberg et al.[85], availability of experimental (human) subjects is limited in practice and. the average number of experimental subjects for an experiment in software engineering is 48.6. This was also a problem in other replication studies, such as [110, 111]. However, all those studies consider only human subjects, not the non-human subjects or artefacts as in our work. Nevertheless, the availability of experimental subjects is also a problem in our replication: In our study, for some systems only later releases were available in online repositories, but not the very old versions used in the original study. We believe that we could get better results from statistical point of view (i.e., fewer differences between the original study and the replication) if we had enough subject systems (and overlapping versions in this study).

4.10 Threats to validity

In this section we present and discuss the threats to validity according to the guidelines by Wohlin et al.[112].

External validity: As in the original study, the first threat is related to using a limited number of systems. We analysed only Apache systems implemented in Java. To reduce this threat, a variety of systems were chosen that vary in application domain, number of versions, size (lines of code) and age. Another threat is that the number of versions analysed per system is different across all systems. However, this is unavoidable because some systems undergo more changes than others. To mitigate this threat, the versions were compared with each other based on version types (major, minor, patch and pre) rather than individual version pairs, i.e., information for each system were compared at the same level of abstraction (version pair type).

Internal validity: An instrumentation effect may occur when differences of the results are caused by the differences in experimental material. We have reduced this threat by performing an exact replication using the same tool as in the original study, the same set of versions from both studies for each software system and an architecture recovery technique used in the original study. Also, we performed different types of comparison (exact comparison, full comparison) using different analysis techniques. All confirmed the same overall findings.

Construct validity: The key threats to construct validity in this study is the accuracy of architecture recovery technique and architecture similarity measures. In order to recover architectures from source code, manually selected .jar files were

considered. Some important .jar files might not have been selected during this manual procedure. To reduce the researcher's bias in manual procedure, we discussed with the authors of the original study and followed the same procedure at our best to select same set of .jar files as in the original study. Furthermore, we used existing metrics that were previously evaluated and used.

Conclusion validity: One threat to conclusion validity is violating the assumptions of statistical tests which may lead to wrong conclusions. In our study, there were few systems with a small sample size (i.e., few versions). To mitigate this threat, when there was a small sample size, we applied a non-parametric test (*Wilcoxon test*) which is robust and has less restrictive assumptions. Also, when results were significantly different according to the t-test, the results were further validated by performing the non-parametric test. Furthermore, all the assumptions for each statistical test (e.g., normality of distribution) were verified before applying a test.

General limitations related to replications: In addition to validity threats, there are general limitations specific to external replications. Two main problems reported by Da Silva et al.[86] are related to presenting replications: 1) the lack of a widely accepted guideline for reporting an experiment replication in software engineering, and 2) the unavailability of lab packages. Our work has been organized based on the proposal of Carver [83] and the related documented experimental packages are also provided as supplementary material[7]. According to Shull et al. [113], documented experimental packages which collect information on experiments make replication more persuasive.

4.11 Conclusions and future works

4.11.1 Conclusions

We replicated an empirical study on architectural changes in open source software systems [27]. This is an external and partially exact replication of the original study. The aim of the replication was to analyse the trends of architecture evolution at system-level and component-level and to gain a deeper understanding of architecture evolution. The replication was performed on the same 14 open source software systems as in the original study. First, a full comparison was performed with all versions of the original study and the replication. We found differences in the value of change metrics values between the two studies. Then,

[7] https://github.com/InfoResearch/Architecture-Decay-An-External-Replication

an exact replication was performed using exactly the same sets of versions from each study for each system. Here, the amount of differences was lower compared to full comparison. The reason for the differences in metrics values in the exact comparison was caused by differences in recovered architectures (input files for metrics calculation) for the same release in the two studies.

The overall findings of this replication are consistent with the results of the original study. Therefore, our replication strengthens the findings of the original study and contributes to the body of knowledge about the architecture evolution of open source software systems. The results of our replication study which further strengthen the results of original study as follows:

- Generally, extensive architectural changes happen between major versions and during the jump from the end of a major version to the next major version.
- The overall trend of extent of architectural changes across the version types of the release sequence is: *Patch* < *Pre* < *Minor* < *MinMaj* < *Major*
- Major architectural changes can also happen in minor system or even in the last moment before a release. This shows that current system's versioning scheme often do not reflect actual architectural changes.

In addition to confirming the insights offered in the original study, our study contributes to understand the nature of architecture stability of systems over a longer period of time. Our study included more releases than the original study, including newer releases that were released over several years after the latest release included in the original study. All these releases including both old and new releases in the replication showed the same overall trend of architectural changes as in the original study. Therefore, our replication also found that architecture evolution does not become more stable over the life time of systems. This finding goes beyond the results of the original study.

Furthermore, our study has important implications for practitioners. As explained in Section 4.1, since we lack empirical data related to architecture evolution that helps practitioners understand *where* and *how* architectural changes occur, our study contribute to expanding such empirical data. Analysing architectural changes not only in system-level but also in component-level indicates that practitioners need to be aware of potential architectural changes that they might not notice since they usually look architectural changes only at the overall system-level. Practitioners can keep track of decay across the evolution path, understand how a

particular maintenance task potentially impacts a system's architecture or how well a proposed change fits with the architecture. This information can be used to reduce unexpected bugs, make more effective decisions when allocating budgets, experts and other resources as explained in Chapter 1. Also, our finding that architectures of open source systems do not stabilize over time indicates that practitioners need to be cautious about potential decay when performing maintenance tasks for long-living systems.

4.11.2 Future work

Our research raises a few interesting points for future works to validate the findings in the original study, but also to improve our understanding of architecture evolution in general. As future works, we can include other open source systems or systems implemented in other programming languages and systems from other application domains (e.g., data warehouse, configuration service, etc.) to enhance the sample size and to increase internal and external validity. Moreover, we could perform some replications on industrial software development projects to analyse the evolution of software projects in industrial context. Finally, we can conduct studies to a) understand reasons for why architectures do not stabilize over time (e.g., major reengineering activities to reduce technical debt), and b) to investigate whether the nature of open source software (many contributors, etc.) makes their architecture more change-prone over time compared to the architectures of commercially developed systems where more rigorous architecture design and planning practices may be in place.

5 Conclusions

This chapter summarizes the thesis and the answers to the research questions outlined in Chapter 1 (Section 5.1). We also reflect on the thesis contributions (Section 5.2) and present directions for future work (Section 5.3).

5.1 Answers to research questions

As explained in Chapter 1, this thesis aims at enhancing our understanding of software maintenance in the context of architecture evolution. Before looking into architecture evolution in more detail, we first aimed at gaining a comprehensive understanding of software maintenance in detail which was the topic of the first research question of this thesis.

5.1.1 RQ1: What is the state-of-the-art of maintenance-related tasks?

To answer RQ1, we conducted a systematic mapping study which was presented in Chapter 3. This mapping study covered the studies published between 2010 and 2017. A total of 55 papers were selected to investigate the state-of-the-art of software maintenance. We refined RQ1 into following two research questions.

RQ1.1: What are the most frequently reported maintenance tasks?
Based on analysing papers that report maintenance tasks, we classified maintenance tasks into nine main types: *bug fixing*, *feature enhancement*, *feature removal*, *feature addition*, *quality improvement*, *pre-change activities*, *post-change activities*, *documentation modification* and *refactoring*. In the last two years of the analysed period, *documentation modification* (e.g., modify code comments, modify design documents) and *pre-change activities* (e.g., locate faulty code element, change impact analysis etc.) were the most reported types. Overall, the most frequently reported maintenance types throughout the study period are *bug fixing* and *refactoring*. These tasks are very high-level and consist of several low-level and more concrete maintenance tasks. Among these low-level tasks, the most reported task is “modifying an algorithm”. Modifying an algorithm includes tasks such as adding break/continue statements, changing data structures, changing mathematical or logical operations, etc.

RQ1.2: How can maintenance tasks be characterized?
We identified seven types of characteristics that can be used to characterize maintenance tasks which were identified in RQ1.1. These characteristics are based on task's *impacted artefact* (e.g., architecture, class, requirement, method etc.), *target* (whether task targets functionality or quality attributes of a system), *impacted stakeholders*, anticipated *complexity* (e.g., number of files/lines changed), *timing* (e.g., design-time, runtime etc.), *tool support* and *frequency* (whether tasks were performed once or frequently). The most frequent way of characterizing tasks in the papers that we analysed is based on whether tasks are tool supported or not, artefact impacted by a task (mostly the architectural impact) and quality attributes the tasks target.

Through a cross-tabulating the findings of RQ1.1 (types of maintenance tasks) and RQ1.2 (characteristics of maintenance tasks), we found that some maintenance tasks are less characterized. In particular, "Change UI" is not well characterized regarding its impact on artefacts, complexity, tool support, etc. Also, only few studies were found that address tool support for maintenance types such as *documentation modification, feature addition/modification* and *post-change activities*. Even though the maintenance type *documentation modification* frequently appeared in the literature recently, little tool support is provided for this type. Furthermore, we found that many tools are already available for *bug fixing* and *refactoring* which can be used by practitioners.

RQ1 provides a comprehensive understanding of software maintenance tasks. Furthermore, characterizing these tasks helps us identifying potential architectural changes. For example, tasks characterized as having component-level impact potentially change the architecture. Based on the characteristics of maintenance tasks, we found that characteristics which are directly related to software architecture (i.e., architectural impact of a task when characterizing based on impacted artefacts) and quality attributes (when characterizing based on target of the task) are frequently used in the literature. This further motivated us to investigate architectural changes in detail. Therefore, we formed the second research question RQ2 of this thesis as below.

5.1.2 RQ2: How do software architectures change during maintenance and evolution?

This research question investigated architectural changes to provide a good understanding of where, when, how and to what extent changes happen. To answer

this research question, we conducted an external replication of a large empirical study of architectural changes which was presented in Chapter 4. This is a replication of an empirical study conducted previously by a group of other researchers to quantify architectural changes across different types of versions of a software system. We analysed the same 14 open source software systems including newer releases using the same tool and metrics introduced in the original study.

RQ2.1: To what extent do architectures change at system-level?
We investigated the extent of architectural changes using different types of version pairs across systems evolution paths. As identified in the original study and confirmed in our replication, substantial changes occur between major version pairs (e.g., 1.0.0, 2.0.0) and during the jump to the next major version from a minor version (e.g., from 1.9.0 to 2.0.0). Smaller changes occur between minor versions (e.g., version pairs 1.1.0 and 1.2.0 or 1.0.0 and 1.1.0) in the same major version. Pre-releases (e.g., version pairs 1.0.0-beta1 and 1.0.0-beta2) which implement user feedback before releasing an official major or minor versions and patch versions (e.g., version pairs 1.2.0 and 1.2.1 or 1.2.1 and 1.2.2) which are usually released after a bug fixing or functionality enhancements impose fewer architectural changes.

RQ2.2: To what extent do architectures change at component-level?
We confirmed that extent of architectural changes at component-level also follows same overall trend across the evolution path as in the changes at system-level (which is explained in RQ2.1). This confirms the finding from the original study. However, we also found that there can be exceptions in the overall trend of architectural changes at both system-level and component-level (as also identified in the original study). Considerably larger architectural changes occur among consecutive minor versions (e.g., 1.3.0 and 1.4.0) which are sometimes similar to or larger than changes during a transition to the next major version from a minor version (e.g., from version 1.8.0 to 2.0.0). Also, larger changes can occur between pre-releases than the changes between minor versions. This indicates that developers do not consider architectural changes in their versioning scheme. In addition to the original study, we found that architectures tend to not stabilize over time and when considering a longer evolution path of systems.

5.2 Thesis contributions and discussion

As outlined in the introduction, the contributions of this thesis are:

- **C1:** A comprehensive taxonomy of maintenance-related tasks with a catalogue of characteristics of tasks (what do tasks impact, when would they be performed, etc.).
- **C2:** A comprehensive understanding of architecture evolution (including evidence that shows where, when, and to what extent architectural changes happen) which is based on a replication of an empirical study that analyses architectural changes of open source software systems.

In Chapter 1, we explained several concrete problems. Below we discuss how the contributions of this thesis and answering the research questions of this thesis address these problems.

- **P1: Difficulty of handling maintenance tasks in general**
  - **P1.1: Incorrect/inconsistent terminology**
    Developers using inconsistent terminology to refer to same task can cause misinterpretation of maintenance tasks (e.g., when allocating budgets, staffing, time etc.). In Chapter 3, we proposed a taxonomy of properly named concrete maintenance tasks which can be used as a common terminology.
  - **P1.2: Lack of sufficient details about maintenance tasks**
    The above taxonomy presents tasks in a hierarchical structure that shows a decomposition of tasks from the most abstract level down to concrete tasks. This provides a good understanding of what sort of lower level tasks need to be performed to complete a particular maintenance task. For example, fixing a bug may involve several different activities such as, changing a return type of a method, adding parameters, changing the UI, etc. In addition to this taxonomy, we also proposed a catalogue of characteristic that can be used by practitioners as a framework to characterize and assess maintenance tasks. Both taxonomy and the catalogue provide a better idea about what it takes to complete the task, the potential impact of a task. For example, knowing whether a task is a change in the implementation of an algorithm or return type, etc., helps allocate more effort and budget for that task.

Above taxonomy and catalogue (contribution C1) that support P1 are based on the current state-of-the-research on software maintenance and offer synthesized knowledge to researchers. Furthermore, insights from the literature could help practitioners benchmark their own practices. The study also proposes future

directions to researchers suggesting to support less addressed maintenance types and gaps in current research landscape that need further attention.

- **P2: Difficulty of handling architecture-related maintenance tasks**
  - **P2.1: Poor understanding of where architecture changes happen**
    In Chapter 4, we showed where architectural changes can happen at system-level and component-level (contribution C2). Practitioners often cannot identify architectural changes at component-level since they only look at system-level architectural changes. Knowing and keep track of component-level changes will help them reduce careless evolution practices and unintentional architecture decay.
  - **P2.2: Poor understanding of how much architecture change happens**
    Chapter 4 also provides extent of architectural changes using two separate metrics for system-level and component-level changes to show when changes happen, and to analyse the nature and trend of architectural changes (contribution C2). Knowing when (e.g., major version, minor version, patch version etc.) changes happen and to what extent and tracking these changes allows practitioners to understand when extensive level or minor architecture changes could happen, how a change impacts a system's architecture, etc. This helps make important decisions such as whether to perform or avoid a maintenance task. This understanding can help reduce unexpected bugs, architecture decay and also predict the cost of change to be performed.

Since we conducted an external replication, it adds credibility to the original study by analysing more versions including additional newer releases that were released over several years after the last release analysed in the original study. This also enriches the body of software engineering knowledge providing insights about architecture evolution. Analysing longer periods of time shows if architectures stabilize over time. Our study showed that architectures do not stabilize over time. Furthermore, this chapter provides reasons why the extent of architectural change differs between the original study and the replication, which will be useful in future research.

5.3 Future work

Based on the findings presented in this thesis we propose below future works.

In Chapter 3, we identified that maintenance types such as post-change activities (verification, validation, updating test etc. in the context of software maintenance) and quality improvement tasks like changing the UI, database, architecture, etc. are less addressed in the literature. Maintenance types such as documentation modification (e.g., updating code comments, Javadocs design/requirements documents etc.), feature addition/modification and post-change activities are less addressed in terms of tool support. Tools can be developed, or existing tools can be improved to support these tasks. Furthermore, based on the characteristics of maintenance tasks identified from the literature, some tasks (e.g., changing the UI, documentation modification etc.) tend to be not characterized in detail. Characteristics such as impacted stakeholders, complexity, timing and frequency are less frequently described in the literature. Researchers can further support these areas and also use the results of this chapter (i.e., the catalogue of characteristic and the taxonomy of maintenance tasks). Both the taxonomy and catalogue can be further enhanced by adding new categories of tasks or characteristics if new types of maintenance tasks or characteristic are identified in future literature reviews.

Chapter 4 raises some future directions to improve our understanding of architecture evolution while strengthening the findings of the original study. We can expand this work to systems implemented in different programming languages and from different application domains and also commercial software systems to increase internal and external validity of the original study. Furthermore, research can be conducted to a) understand the reasons why systems architectures do not stabilize overtime, and b) to explore whether the nature of open source systems (e.g., many contributors, more flexible due to no schedules or explicit system-level design, works not assigned and developers undertake what they choose etc.) has an impact on system architectures to be more change-prone than architectures of commercially developed systems.

Overall, this thesis aims at improving our understanding of software maintenance in the context of architecture evolution. To further support our overall goal, we can explore approaches that support predicting the extent of architectural change of a particular maintenance task based on historically recorded data (e.g., through mining software repositories). Furthermore, we can explore approaches to evaluate architecture decay throughout the evolution path. To evaluate architecture decay, we

can first explore existing tools to detect symptoms of decay or architecture smells. For example, there are different types of architectural smells (e.g., coupling-based smells, dependency-based smells etc.) [114]. We can explore metrics to measure different types of architectural smell as an indication of decay. Here, a combination of metrics may provide more reliable identification of decay. Furthermore, we can integrate a set of refactoring techniques that support removing each type of smell.

Decay can be further evaluated by characterizing its impact on system quality attributes which also can be used to measure sustainability of a system. Therefore, we hope to understand what architecture smells affect what quality attributes and identify a relationship between architecture smells and quality attributes. For example, we can use results of a preliminary stage study by Le et al. [115] which relates architecture smell, metrics to measure smell and their impacted quality attributes. According to Venters et al. [6], evaluating the sustainability of a system is an emerging work and requires more research to evaluate sustainability and to identify the most appropriate architectural-level metrics to analyse sustainability of software architectures.

In addition, we can explore the possibility of developing or assessing software visualization tools that visualize and monitor architecture changes to help developers (rather than architects) comprehend architecture evolution. Such tools could also help developers understand the impact of lower level implementation tasks on architecture evolution. This would create awareness amongst developers for architecture change at component-level (which, as our study showed, can be hidden by architecture change at system-level which is often controlled by architects). Furthermore, these tools can support recommending a set of best practices for a particular change in the system to control architecture decay.

References

[1] M. Riaz, M. Sulayman, and H. Naqvi, "Architectural Decay during Continuous Software Evolution and Impact of 'Design for Change' on Software Architecture," in *International Conference on Advanced Software Engineering and Its Applications (ASEA)*, 2009, pp. 119-126: Springer.

[2] B. J. Williams and J. C. Carver, "Characterizing software architecture changes: A systematic review," *Information and Software Technology*, vol. 52, no. 1, pp. 31-51, 2010.

[3] S. Black, "Computing ripple effect forsoftware maintenance," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 4, pp. 263-279, 2001.

[4] P. Grubb and A. A. Takang, *Software Maintenance : Concepts and Practice*, 2nd ed. World Scientific Publishing Co Pte Ltd, Singapore, 2014.

[5] R. L. Glass, "Learning to distinguish a solution from a problem," *IEEE Software*, vol. 21, no. 3, pp. 111 - 112, 2004.

[6] C. C. Venters *et al.*, "Software sustainability: Research and practice from a software architecture viewpoint," *Journal of Systems and Software*, vol. 138, pp. 174-188, 2018.

[7] J. Král and M. Žemlička, "Simplifying maintenance by application of architectural services," in *14th International Conference on Computational Science and Its Applications (ICCSA)*, 2014, pp. 476-491: Springer.

[8] X. Sun, B. Li, H. Leung, B. Li, and Y. Li, "MSR4SM: Using topic models to effectively mining software repositories for software maintenance tasks," *Information and Software Technology*, vol. 66, pp. 1-12, 2015.

[9] P. Behnamghader, D. M. Le, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "A Large-Scale Study of Architectural Evolution in Open-Source Software Systems," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1146 - 1193, 2017.

[10] J. Garcia, I. Krka, N. Medvidovic, and C. Douglas, "A Framework for Obtaining the Ground-Truth in Architectural Recovery," in *10th Joint Working IEEE/IFIP Conference on Software Architecture and 6th European Conference on Software Architecture (WICSA / ECSA)*, 2012, pp. 292-296: IEEE.

[11] J. Garcia, I. Ivkovic, and N. Medvidovic, "A Comparative Analysis of Software Architecture Recovery Techniques," in *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 486-496: IEEE.

[12] Y. Ku, J. Du, Y. Yang, and Q. Wang, "Estimating Software Maintenance Effort from Use Cases: an Industrial Case Study," in *27th International Conference on Software Maintenance (ICSM)*, 2011, pp. 482-491: IEEE.

[13] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *35th International Conference on Software Engineering (ICSE)*, 2013, pp. 392-401: IEEE.

[14] M. Usman, E. Mendes, and J. Börstler, "Effort Estimation in Agile Software Development: A Survey on the State of the Practice," in *19th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2015, pp. 1-10: ACM.

[15] B. Govin, N. Anquetil, A. Etien, S. Ducasse, and A. Monegier, "How Can We Help Software Rearchitecting Efforts? Study of an Industrial Case," in *32nd International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 509-518: IEEE.

[16] T. Dyba and T. Dingsøyr, "Empirical studies of agile software development: A systematic review," *Information and Software Technology*, vol. 50, no. 9, pp. 833-859, 2008.

[17] G. Bavota and B. Russo, "A Large-Scale Empirical Study on Self-Admitted Technical Debt," in *13th International Conference on Mining Software Repositories (MSR)*, 2016, pp. 315-326: ACM.

[18] R. Coelho, L. Almeida, G. Gousios, A. v. Deursen, and C. Treude, "Exception handling bug hazards in Android - Results from a mining study and an exploratory survey," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1264–1304, 2017.

[19] E. Lim, N. Taksande, and C. B. Seaman, "A balancing act:What software practitioners have to say about technical debt," *IEEE Software*, vol. 29, no. 6, pp. 22-27, 2012.

[20] D. Falessi, P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt at the crossroads of research and practice: report on the fifth international workshop on managing technical debt," *SIGSOFT Software Engineering Notes*, vol. 39, no. 2, pp. 31-33, 2014.

[21] M. M. Lehman and L. A. Belady, *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985, p. 538.

[22] R. Mo, J. Garcia, Y. Cai, and N. Medvidovic, "Mapping architectural decay instances to dependency models," in *4th International Workshop on Managing Technical Debt (MTD)*, 2013, pp. 39-46: IEEE.

[23] N. Ramasubbu and C. F. Kemerer, "Technical Debt and the Reliability of Enterprise Software Systems: A Competing Risks Analysis," *Management Science*, vol. 62, no. 5, pp. 1487-1510, 2016.

[24] T. Besker, A. Martini, and J. Bosch, "Impact of Architectural Technical Debt on Daily Software Development Work — A Survey of Software Practitioners," in *43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2017, pp. 278-287: IEEE.

[25] T. Besker, A. Martini, and J. Bosch, "Managing architectural technical debt: A unified model and systematic literature review," *Journal of Systems and Software*, vol. 135, pp. 1-16, 2018.

[26] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *38th International Conference on Software Engineering (ICSE)*, 2016, pp. 488-498: IEEE.
[27] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "An Empirical Study of Architectural Change in Open-Source Software Systems," in *12th Working Conference on Mining Software Repositories (MSR)*, 2015, pp. 235-245: IEEE.
[28] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Information and Software Technology*, vol. 64, pp. 1-18, 2015.
[29] B. Kitchenham and S. Charters, "Guidelines for Performing Systematic Literature Reviews in Software Engineering,Technical report," 2007.
[30] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," in *12th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2008, pp. 68-77: ACM.
[31] N. Juristo and S. Vegas, "Using differences among replications of software engineering experiments to gain knowledge," in *3rd International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2009, pp. 356-366: IEEE.
[32] O. S. Gómez, N. Juristo, and S. Vegas, "Understanding replication of experiments in software engineering: A classification," *Information and Software Technology*, vol. 56, no. 8, pp. 1033-1048, 2014.
[33] O. S. Gómez, N. Juristo, and S. Vegas, "Replications Types in Experimental Disciplines," in *4th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2010, pp. 1-10: ACM.
[34] F. J.Shull, J. C. Carver, S. Vegas, and N. Juristo, "The role of replications in Empirical Software Engineering," *Empirical Software Engineering*, vol. 13, no. 2, pp. 211-218, 2008.
[35] K. Bennett and V. Rajlich, "Software maintenance and evolution: a roadmap," in *22nd International Conference on Software Engineering (ICSE)*, 2000, pp. 73-87: ACM.
[36] *ISO/IEC 14764:2006 IEEE Standard 14764:2006*, 2006.
[37] *IEEE Std 1219-1998: IEEE Standard for Software Maintenance*, 1998.
[38] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40-52, 1992.
[39] P. Kruchten, "Documentation of Software Architecture from a Knowledge Management Perspective – Design Representation," in *Software Architecture Knowledge Management: Theory and Practice*: Springer, Berlin, Heidelberg, 2009, pp. 39-57.
[40] P. Kruchten, H. Obbink, and J. Stafford, "The Past, Present, and Future for Software Architecture," *IEEE Software*, vol. 23, no. 2, pp. 22-30, 2006.
[41] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Addison-Wesley Professional, 2003.

[42] R. Capilla, A. Jansen, A. Tang, P. Avgeriou, and M. A. Babar, "10 years of software architecture knowledge management: Practice and future," *The Journal of Systems and Software,* vol. 116, pp. 191-205, 2016.
[43] "ISO/IEC/IEEE Systems and software engineering -- Architecture description," *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pp. 1-46, 2011.
[44] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions," in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2005, pp. 109-120: IEEE.
[45] *ANSI/ IEEE Std 729-1983: IEEE Standard Glossary of Software Engineering Terminology*. IEEE, 1983.
[46] P. Bourque and R. E. Fairley, *Guide to the Software Engineering Body of Knowledge (SWEBOK(R))*, 3rd ed. IEEE 2014.
[47] P. Kruchten, P. Lago, and H. van Vliet, "Building Up and Reasoning About Architectural Knowledge.," in *Quality of Software Architectures*: Springer, Berlin, Heidelberg, 2006, pp. 43-58.
[48] N. Medvidovic and R. Taylor, "Software architecture: foundations, theory, and practice," in *32nd International Conference on software engineering (ICSE)*, 2010, pp. 471-472: ACM.
[49] R. N. Taylor, N. Medvidović, and E. M. Dashofy, *Software architecture: foundations, theory, and practice*. Hoboken, NJ: John Wiley, 2010.
[50] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Transactions on Software Engineering,* vol. 27, no. 1, pp. 1-12, 2001.
[51] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software,* vol. 85, no. 1, pp. 132-151, 2012.
[52] L. Hochstein and M. Lindvall, "Combating architectural degeneration: a survey," *Information and Software Technology,* vol. 47, no. 10, pp. 643-656, 2005.
[53] J. van Gurp and J. Bosch, "Design erosion: problems and causes," *The Journal of Systems and Software,* vol. 61, no. 2, pp. 105-119, 2002.
[54] C. Izurieta and J. M. Bieman, "How software designs decay: A pilot study of pattern evolution," in *1st International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2007, pp. 449-451: IEEE.
[55] R. L. Novais, A. Torres, T. S. Mendes, M. Mendonça, and N. Zazworka, "Software evolution visualization: A systematic mapping study," *Information and Software Technology,* vol. 55, no. 11, pp. 1860-1883, 2013.
[56] J. Saraiva *et al.*, "Aspect-oriented software maintenance metrics: A systematic mapping study," in *16th International Conference on Evaluation & Assessment in Software Engineering (EASE)*, 2012, pp. 253-262: IET.
[57] M. Riaz, E. Mendes, and E. Tempero, "A Systematic Review of Software Maintainability Prediction and Metrics," in *3rd International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2009: IEEE.

[58] H. C. Benestad, B. Anda, and E. Arisholm, "Understanding software maintenance and evolution by analyzing individual changes: A literature review," *Journal of Software Maintenance and Evolution,* vol. 21, no. 6, pp. 349-378, 2009.

[59] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering – A systematic literature review," *Information and Software Technology,* vol. 51, no. 1, pp. 7-15, 2009.

[60] B. Kitchenham *et al.*, "Systematic literature reviews in software engineering – A tertiary study," *Information and Software Technology,* vol. 52, no. 8, pp. 792-805, 2010.

[61] F. Q. B. da Silva, A. L. M. Santos, S. Soares, A. C. C. França, C. V. F. Monteiro, and F. F. Maciel, "Six years of systematic literature reviews in software engineering: An updated tertiary study," *Information and Software Technology,* vol. 53, no. 9, pp. 899-913, 2011.

[62] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert Systems with Applications,* vol. 36, no. 4, pp. 7346-7354, 2009.

[63] P. Runeson *et al.*, "What Do We Know about Defect Detection Methods? [software testing]," *IEEE Software,* vol. 23, no. 3, pp. 82-90, 2006.

[64] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis," *IEEE Transactions on Software Engineering,* vol. 35, no. 5, pp. 684-702, 2009.

[65] W. Hordijk, M. L. Ponisio, and R. Wieringa, "Harmfulness of Code Duplication-A Structured Review of the Evidence," in *13th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2009, pp. 1-10.

[66] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 19, no. 2, pp. 77-131, 2007.

[67] B. P. Lientz and E. B. Swanson, "Software maintenance management: a study of the maintenance of computer applications software in 487 data processing organizations," ed: Addison-Wesley, Reading, MA, 1980.

[68] I. H. Lin and D. A. Gustafson, "Classifying Software Maintenance," in *Conference on Software Maintenance*, 1988, pp. 241-247: IEEE

[69] Chapin, "Software Maintenance Types-A Fresh View," in *International Conference on Software Maintenance (ICSM)*, 2000, pp. 247-252: IEEE.

[70] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniese, "Towards a taxonomy of software change," *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 17, no. 5, pp. 309-332, 2005.

[71] M. Elkholy and A. Elfatatry, "Change Taxonomy: A Fine-Grained Classification of Software Change," *IT Professional,* vol. 20, no. 4, pp. 28-36, 2018.

[72] L. Chen, M. Ali Babar, and H. Zhang, "Towards an evidence-based understanding of electronic data sources," in *14th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2010, pp. 135-138: BCS Learning & Development Ltd.

[73] M. Kuhrmann, D. M. Fernandez, and M. Daneva, "On the pragmatic design of literature studies in software engineering: an experience-based guideline," *Empirical Software Engineering*, vol. 22, no. 6, pp. 2852-2891, 2017.

[74] H. Zhang, M. A. Babar, and P. Tell, "Identifying relevant studies in software engineering," *Information and Software Technology*, vol. 53, no. 6, pp. 625-637, 2011.

[75] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126-139, 2004.

[76] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W. G. Tan, "Types of software evolution and software maintenance," *Journal of Software Maintenance and Evolution: Research And Practice*, vol. 13, no. 1, pp. 3-30, 2001.

[77] J. Zhi, V. Garousi-Yusifoğlu, B. Sun, G. Garousi, S. Shahnewaz, and G. Ruhe, "Cost, benefits and quality of software development documentation: A systematic mapping," *Journal of Systems and Software*, vol. 99, pp. 175-198, 2015.

[78] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley, Boston, 1999.

[79] S. Ducasse and D. Pollet, "Software Architecture Reconstruction: A Process-Oriented Taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573-591, 2009.

[80] A. Murgia, G. Concas, S. Pinna, R. Tonelli, and I. Turnu, "Empirical study of software quality evolution in open source projects using agile practices," in *1st International Symposium on Emerging Trends in Software Metrics (ETSM)*, 2009, pp. 1-12: Michele Marchesi.

[81] A. Chatzigeorgiou and A. Manakos, "Investigating the Evolution of Bad Smells in Object-Oriented Code," in *7th International Conference on the Quality of Information and Communications Technology (QUATIC)*, 2010, pp. 106-115: IEEE.

[82] S. Hassaine, Y.-G. Gu´eh´eneuc§, S. Hamel, and G. Antoniol, "ADvISE: Architectural Decay In Software Evolution," in *16th European Conference on Software Maintenance and Reengineering (CSMR)*, 2012, pp. 267-276: IEEE.

[83] J. C. Carver, "Towards reporting guidelines for experimental replications: A proposal," in *1st International Workshop on Replication in Empirical Software Engineering Research (RESER)*, 2010, pp. 1-4: ACM.

[84] F. Ricca, G. Scanniello, M. Torchiano, G. Reggio, and E. Astesiano, "On the Effectiveness of Screen Mockups in Requirements Engineering: Results from an Internal Replication," in *4th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2010, pp. 1-10: ACM.

[85] D. I. K. Sjøberg *et al.*, "A Survey of Controlled Experiments in Software Engineering," *IEEE Transactions on Software Engineering*, vol. 31, no. 9, pp. 733-753, 2005.

[86] F. Q. B. d. Silva *et al.*, "Replication of empirical studies in software engineering research: a systematic mapping study," *Empirical Software Engineering*, vol. 19, no. 3, pp. 501-557, 2014.

[87] M. T. Baldassarre, J. Carver, and N. Juristo, "Replication types: towards a shared taxonomy," in *18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2014, pp. 1-4: ACM.

[88] J. C. Carver, N. Juristo, M. T. Baldassarre, and S. Vegas, "Replications of software engineering experiments," *Empirical Software Engineering*, vol. 19, no. 2, pp. 267-276, 2014.

[89] N. Juristo, S. Vegas, M. Solari, S. Abrahão, and I. Ramos, "A process for managing interaction between experimenters to get useful similar replications," *Information and Software Technology*, vol. 55, no. 2, pp. 215-225, 2013.

[90] C. V. C. de Magalhães, F. Q. B. da Silva, R. E. S. Santos, and M. Suassuna, "Investigations about replication of empirical studies in software engineering: A systematic mapping study," *Information and Software Technology*, vol. 64, pp. 76-101, 2015.

[91] J. Siegmund, N. Siegmund, and S. Apel, "Views on Internal and External Validity in Empirical Software Engineering," in *37th International Conference on Software Engineering (ICSE)*, 2015, pp. 9-19: IEEE.

[92] M. Galster and D. Weyns, "Empirical Research in Software Architecture: How Far have We Come?." in *13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2016, pp. 11-20: IEEE.

[93] T. Lutellier, D. Chollak, and J. Garcia, "Comparing Software Architecture Recovery Techniques Using Accurate Dependencies," in *37th International Conference on Software Engineering (ICSE)*, 2015, pp. 69-78: IEEE.

[94] M. Zahid, Z. Mehmmod, and I. Inayat, "Evolution in Software Architecture Recovery Techniques – A Survey," in *13th International Conference on Emerging Technologies (ICET)*, 2017, pp. 1-6: IEEE.

[95] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *6th International Workshop on Program Comprehension (IWPC)*, 1998, pp. 45-52: IEEE.

[96] K. Kobayashi, M. Kamimura, K. Kato, K. Yano, and A. Matsuo, "Feature-gathering dependency-based software clustering using Dedication and Modularity," in *28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 462-471: IEEE.

[97] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the Bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193-208, 2006.

[98] K. Praditwong, M. Harman, and X. Yao, "Software Module Clustering as a Multi-Objective Search Problem," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 264-282, 2011.
[99] O. Maqbool and H. A. Babri, "Hierarchical Clustering for Software Architecture Recovery," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759-780, 2007.
[100] R. Koschke, *Architecture Reconstruction.* (Lecture Notes in Computer Science). Springer-Verlag Berlin, Heidelberg, 2009, pp. 140-173.
[101] C. Raibulet and L. Masciadri, "Evaluation of Dynamic Adaptivity Through Metrics:an Achievable Target?," in *Joint 8th Working IEEE/IFIP Conference on Software Architecture and 3rd European Conference on Software Architecture (WICSA/ECSA)*, 2009, pp. 341-344: IEEE.
[102] T. Nakamura and V. R. Basili, "Metrics of Software Architecture Changes Based on Structural Distance," in *11th International Software Metrics Symposium (METRICS)*, 2005, pp. 24-34: IEEE.
[103] D. Reimanis, C. Izurieta, R. Luhr, L. Xiao, Y. Cai, and G. Rudy, "A Replication Case Study to Measure the Architectural Quality of a Commercial System," in *8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2014, pp. 1-8: ACM.
[104] B. Rossi and B. Russo, "Evolution of design patterns: a replication study," in *8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2014, pp. 1-4: ACM.
[105] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing Architectural Recovery Using Concerns," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2011, pp. 552-555: IEEE.
[106] D. M. Blei. (2012) Probabilistic topic models. *Communications of the ACM* [Review]. 77-84.
[107] V. Tzerpos and R. Holt, "ACDC: An algorithm for comprehension-driven clustering," in *7th Working Conference on Reverse Engineering (WCRE)*, 2000, pp. 258-267: IEEE.
[108] B. Agnew, C. Hofmeister, and J. Purtilo, "Planning for change: a reconfiguration language for distributed systems," *Distributed Systems Engineering*, vol. 1, no. 5, pp. 313-322, 1994.
[109] N. Medvidovic, "ADLs and dynamic architecture changes," in *Joint 2nd International Software Architecture Workshop (ISAW) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 workshops*, 1996, pp. 24-27: ACM.
[110] C. Apa, O. Dieste, G. Edison, G. Espinosa, R. Efraín, and C. Fonseca, "Effectiveness for detecting faults within and outside the scope of testing techniques: an independent replication," *Empirical Software Engineering*, vol. 19, no. 2, pp. 378-417, 2014.

[111] F. Calefato, F. Lanubile, T. Conte, and R. Prikladnicki, "Assessing the Impact of Real-Time Machine Translation on Requirements Meetings: A Replicated Experiment," in *ACM/IEEE 6th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2012, pp. 251-260: ACM.

[112] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer, Berlin, Heidelberg, 2012.

[113] F. Shull, V. Basili, J. Carver, and J. C. Maldonado, "Replicating Software Engineering Experiments: Addressing the Tacit Knowledge Problem," in *International Symposium on Empirical Software Engineering (ISESE)*, 2002, pp. 7-16: IEEE.

[114] D. Le, D. Link, Y. Zhao, A. Shahbazian, C. Mattmann, and N. Medvidovic, "Toward a Classification Framework for Software Architectural Smells," *Techincal Report csse. usc. edu*, 2017.

[115] D. M. Le, C. Carrillo, R. Capilla, and N. Medvidovic, "Relating Architectural Decay and Sustainability of Software Systems," in *13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2016, pp. 178-181: IEEE.

Appendix

Appendix A

Publication venues

IEEE Transactions on Software Engineering
Journal of Systems and Software
Information and Software Technology
ACM Transactions on Software Engineering and Methodology
Empirical Software Engineering
Studies in Computational Intelligence
Journal of Software: Evolution and Process
Software Testing, Verification and Reliability
Software Quality Journal
Advanced Information Systems Engineering
Automated Software Engineering
International Symposium on the Foundations of Software Engineering
International Conference on Software Engineering
International Conference on Predictive Models in Software Engineering
Working Conference on Mining Software Repositories
International Conference on Program Comprehension
Working IEEE/IFIP Conference on Software Architecture
International Conference on Automated Software Engineering
IEEE International Conference on Web Services
Working Conference on Reverse Engineering
IEEE International Conference on Software Maintenance
European Conference on Software Maintenance and Reengineering
Asia-Pacific Software Engineering Conference
International Systems and Software Product Line Conference
International Symposium on Empirical Software Engineering and Measurement
International Conference on Software Maintenance and Evolution
Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering
International Symposium on Software Reliability Engineering
International Conference on Software Engineering and Knowledge Engineering
International Conference on Computer Science and Information Technology
European Conference on Object-Oriented Programming
European Conference on Software Architecture
International Conference on Fundamental Approaches to Software Engineering
International Conference on Computational Science and Its Applications
International Conference on Agile Software Development
National Software Application Conference

Appendix B

Selected papers for systematic mapping study

Papers marked with '*' are included in the quasi-gold standard.

[P1] Murphy-Hill, E., Zimmermann, T., Bird, C. and Nagappan, N., 2015. The design space of bug fixes and how developers navigate it. *IEEE Transactions on Software Engineering 41*, 1, 65-81.

[P2] Nguyen, A. T., Nguyen, T. T., Nguyen, H. A. and Nguyen, T. N., 2012. Multi-layered approach for recovering links between bug reports and fixes. In *International Symposium on the Foundations of Software Engineering*, 1-11.

[P3] * Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J. and Nguyen, T. N., 2010. Recurring bug fixes in object-oriented programs. In *International Conference on Software Engineering*, 315-324.

[P4] * Gethers, M., Dit, B., Kagdi, H. and Poshyvanyk, D., 2012. Integrated impact analysis for managing software changes. In *34th International Conference on Software Engineering*, 430-440.

[P5] Bachmann, A., Bird, C., Rahman, F., Devanbu, P. and Bernstein, A., 2010. The missing links: bugs and bug-fix commits. In *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering*, 97-106.

[P6] Murgia, A., Concas, G., Tonelli, R., Ortu, M., Demeyer, S. and Marchesi, M., 2014. On the influence of maintenance activity types on the issue resolution time. In *Proceedings of the 10th International Conference on Predictive Models in Software Engineering*, 12-21.

[P7] Zaman, S., Adams, B. and Hassan, A. E., 2011. Security versus performance bugs: a case study on Firefox. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, 93-102.

[P8] Palomba, F., Zaidman, A., Oliveto, R. and Lucia, A. D., 2017. An Exploratory Study on the Relationship between Changes and Refactoring. In *25th International Conference on Program Comprehension*, 176-185.

[P9] * Bavota, G. and Russo, B., 2016. A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th International Conference on Mining Software Repositories*, 315-326.

[P10] * Zhong, H. and Su, Z., 2015. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, 913-923.

[P11] Dragomir, A., Harun, M. F. and Lichter, H., 2014. On bridging the gap between practice and vision for software architecture reconstruction and evolution: a toolbox perspective. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture Companion Volume*, 1-4.

[P12] * Ray, B., Nagappan, M., Bird, C., Nagappan, N. and Zimmermann, T., 2015. The uniqueness of changes: characteristics and applications. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, 34-44.

[P13] * Ma, W., Chen, L., Zhang, X., Zhou, Y. and Xu, B., How do developers fix cross-project correlated bugs? a case study on the GitHub scientific python ecosystem. In *Proceedings of the 39th International Conference on software engineering*, 381-392.

[P14] Moha, N., Gueheneuc, Y., Duchien, L. and Meur, A. L., 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering 36*, 1, 20-36.

[P15] Gao, Q., Zhang, H., Wang, J., Xiong, Y., Zhang, L. and Mei, H., 2015. Fixing Recurring Crash Bugs via Analyzing Q&A Sites (T). In *30th International Conference on Automated Software Engineering*, 307-318.

[P16] Fokaefs, M., Mikhaiel, R., Tsantalis, N., Stroulia, E. and Lau, A., 2011. An Empirical Study on Web Service Evolution. In *2011 IEEE International Conference on Web Services*, 49-56.

[P17] * Thung, F., Lo, D. and Jiang, L., 2012. Automatic Defect Categorization. In *19th Working Conference on Reverse Engineering*, 205-214.

[P18] * Linares-Vásquez, M., Hossen, K., Dang, H., Kagdi, H., Gethers, M. and Poshyvanyk, D., 2012. Triaging incoming change requests: Bug or commit history, or code authorship? In *28th IEEE International Conference on Software Maintenance*, 451-460.

[P19] Jamshidi, P., Ghafari, M., Ahmad, A. and Pahl, C., 2013. A Framework for Classifying and Comparing Architecture-centric Software Evolution Research. In *17th European Conference on Software Maintenance and Reengineering*, 305-314.

[P20] * Maia, M. d. A. and Lafetá, R. F., 2013. On the impact of trace-based feature location in the performance of software maintainers. *Journal of Systems and Software 86*, 4, 1023-1037.

[P21] Jayatilleke, S. and Lai, R., 2018. A systematic review of requirements change management. *Information and Software Technology 93*, 163-185.

[P22] * Nishizono, K., Morisakl, S., Vivanco, R. and Matsumoto, K., 2011. Source code comprehension strategies and metrics to predict comprehension effort in software maintenance and evolution tasks - an empirical study with industry practitioners. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, 473-481.

[P23] Besker, T., Martini, A. and Bosch, J., 2018. Managing architectural technical debt: A unified model and systematic literature review. *Journal of Systems and Software 135*, 1-16.

[P24] Amanatidis, T., Chatzigeorgiou, A. and Ampatzoglou, A., 2017. The relation between technical debt and corrective maintenance in PHP web applications. *Information and Software Technology 90*, 70-74.

[P25] Martini, A., Besker, T. and Bosch, J., 2016. The Introduction of Technical Debt Tracking in Large Companies. In *Proceedings of the 23rd Asia-Pacific Software Engineering Conference*. 161-168.

[P26] Thüm, T., Ribeiro, M., Schröter, R., Siegmund, J. and Dalton, F., 2016. Product-line maintenance with emergent contract interfaces. In *Proceedings of the Proceedings of the 20th International Systems and Software Product Line Conference*, 134-143.

[P27] Hira, A. and Boehm, B., 2016. Using Software Non-Functional Assessment Process to Complement Function Points for Software Maintenance. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 1-6.

[P28] Miura, K., McIntosh, S., Kamei, Y., Hassan, A. E. and Ubayashi, N., 2016. The Impact of Task Granularity on Co-evolution Analyses. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 1-10.

[P29] Candela, I., Bavota, G., Russo, B. and Oliveto, R., 2016. Using Cohesion and Coupling for Software Remodularization: Is It Enough? *ACM Transactions on Software Engineering and Methodology 25*, 3, 1-28.

[P30] Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K. and Deb, K., 2016. Multi-Criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study. *ACM Transactions on Software Engineering and Methodology 25*, 3, 1-53.

[P31] Heeager, L. T. and Rose, J., 2015. Optimising agile development practices for the maintenance operation: nine heuristics. *Empirical Software Engineering 20*, 6, 1762-1784.

[P32] * Szőke, G., Nagy, C., Hegedűs, P., Ferenc, R. and Gyimóthy, T., 2015. Do automatic refactorings improve maintainability? An industrial case study. In *IEEE International Conference on Software Maintenance and Evolution*, 429-438.

[P33] Sun, X., Li, B., Leung, H., Li, B. and Li, Y., 2015. MSR4SM: Using topic models to effectively mining software repositories for software maintenance tasks. *Information and Software Technology 66*, 1-12.

[P34] Kevic, K., Walters, B. M., Shaffer, T. R., Sharif, B., Shepherd, D. C. and Fritz, T., 2015. Tracing software developers' eyes and interactions for change tasks. In *Joint Meeting on Foundations of Software Engineering*, 202-213.

[P35] Sun, X., Li, B., Li, Y. and Chen, Y., 2015. What information in software historical repositories do we need to support software maintenance tasks? an approach based on topic model. *Studies in Computational Intelligence 566*, 27-37.

[P36] Fernández-Sáez, A. M., Genero, M., Chaudron, M. R. V., Caivano, D. and Ramos, I., 2015. Are Forward Designed or Reverse-Engineered UML diagrams more helpful for code maintenance?: A family of experiments. *Information and Software Technology 57*, 644-663.

[P37] Fritz, T., Shepherd, D. C., Kevic, K., Snipes, W. and Bräunlich, C., 2014. Developers' code context models for change tasks. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 7-18.

[P38] * Sun, X., Leung, H., Li, B. and Li, B., 2014. Change impact analysis and changeability assessment for a change proposal: An empirical study. *Journal of Systems and Software 96*, 51-60.

[P39] Ahmad, A., Jamshidi, P. and Pahl, C., 2014. Classification and comparison of architecture evolution reuse knowledge-a systematic review. *Journal of Software: Evolution and Process 26*, 7, 654-691.

[P40] Yang, H., Wang, C., Shi, Q., Feng, Y. and Chen, Z. Bug inducing analysis to prevent fault prone bug fixes. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*. 620-625.

[P41] Li, B., Sun, X., Leung, H. and Zhang, S., 2013. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability 23*, 8, 613-646.

[P42] Nguyen, H. A., Nguyen, A. T. and Nguyen, T. N., 2013. Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. In *International Symposium on Software Reliability Engineering* 138-147.

[P43] * Jaafar, F., Hassaine, S., Guéhéneuc, Y., Hamel, S. and Adams, B., 2013. On the Relationship between Program Evolution and Fault-Proneness: An Empirical Study. In *European Conference on Software Maintenance and Reengineering*, 15-24.

[P44] Lagerström, R., Johnson, P. and Ekstedt, M., 2010. Architecture analysis of enterprise systems modifiability: a metamodel for software change cost estimation. *Software Quality Journal 18*, 4, 437-468.

[P45] Tran, L. M. S. and Massacci, F., 2011. Dealing with known unknowns: Towards a game-theoretic foundation for software requirement evolution. In *International Conference on Advanced Information Systems Engineering*, 62-76.

[P46] Kavitha, D. and Sheshasaayee, A., 2012. Requirements volatility in software maintenance. In *International Conference on Computer Science and Information Technology*, 142-150.

[P47] Negara, S., Chen, N., Vakilian, M., Johnson, R. E. and Dig, D., 2013. A comparative study of manual and automated refactorings. In *European Conference on Object-Oriented Programming*, 552-576.

[P48] Ouni, A., Kessentini, M., Sahraoui, H. and Boukadoum, M., 2013. Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering 20*, 1, 47-79.

[P49] Herold, S. and Mair, M., 2014. Recommending refactorings to re-establish architectural consistency. In *European Conference on Software Architecture*, 390-397.

[P50] Mahouachi, R., Kessentini, M. and Ghedira, K., 2012. A new design defects classification: Marrying detection and correction. In *International Conference on Fundamental Approaches to Software Engineering*, 455-470.

[P51] Szőke, G., Nagy, C., Ferenc, R. and Gyimóthy, T., 2014. A Case Study of Refactoring Large-Scale Industrial Systems to Efficiently Improve [Source Code Quality. In *International Conference on Computational Science and Its Applications*, 524-540.

[P52] Counsell, S., Swift, S., Murgia, A., Tonelli, R., Marchesi, M. and Concas, G., 2014. Are some refactorings attached to fault-prone classes and others to fault-free classes? In *International Conference on Agile Software Development*. 136-147.

[P53] Mäder, P. and Egyed, A., 2015. Do developers benefit from requirements traceability when evolving and maintaining a software system? *Empirical Software Engineering 20*, 2, 413-441.

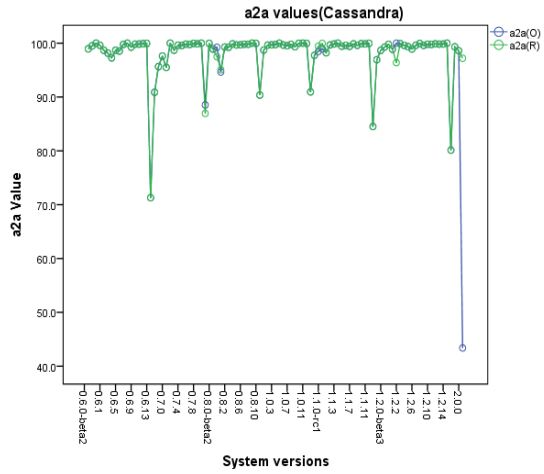
[P54] Ricca, F., Torchiano, M., Leotta, M., Tiso, A., Guerrini, G. and Reggio, G., 2018. On the impact of state-based model-driven development on maintainability: a family of experiments using UniMod. *Empirical Software Engineering 23*, 3, 1743-1790.

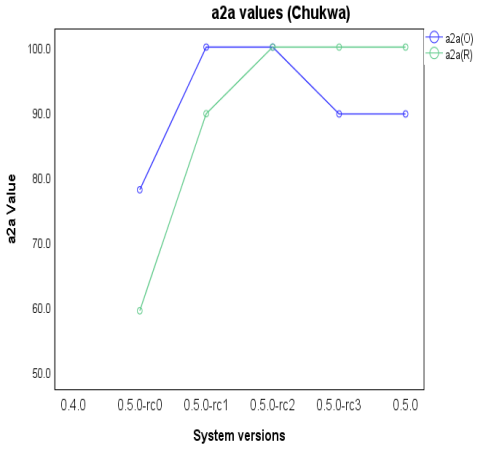
[P55] Kuang, H., Nie, J., Hu, H. and Lü, J., 2016. Improving Automatic Identification of Outdated Requirements by Using Closeness Analysis Based on Source Code Changes. In *National Software Application Conference*, 52-67.

Appendix C

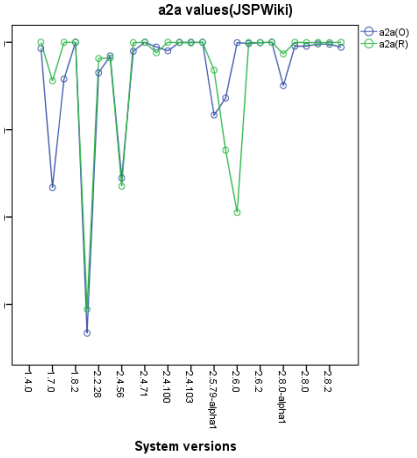
System	Common versions in original study and replication	% of common versions
ActiveMQ	Different set of versions in replication	-
Cassandra	0.6.0-beta2, 0.6.0-beta3, 0.6.0-rc1, 0.6.0, 0.6.1, 0.6.2, 0.6.3, 0.6.4, 0.6.5, 0.6.6, 0.6.7, 0.6.8, 0.6.9, 0.6.10, 0.6.11, 0.6.12, 0.6.13 0.7.0-beta1, 0.7.0-beta2, 0.7.0-beta3,0.7.0, 0.7.1, 0.7.2, 0.7.3, 0.7.4, 0.7.5, 0.7.6, 0.7.7, 0.7.8, 0.7.9, 0.7.10, 0.8.0-beta1, 0.8.0-beta2, 0.8.0-rc1, 0.8.0, 0.8.1, 0.8.2, 0.8.3, 0.8.4, 0.8.5, 0.8.6, 0.8.7, 0.8.8, 0.8.9, 0.8.10, 1.0.0, 1.0.1, 1.0.2, 1.0.3, 1.0.4, 1.0.5, 1.0.6, 1.0.7, 1.0.8, 1.0.9, 1.0.10, 1.0.11, 1.0.12, 1.1.0-beta1, 1.1.0-beta2, 1.1.0-rc1, 1.1.0, 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, 1.1.6, 1.1.7, 1.1.8, 1.1.9, 1.1.10, 1.1.11, 1.1.12,1.2.0-beta1, 1.2.0-beta2, 1.2.0-beta3, 1.2.0-rc1, 1.2.0-rc2, 1.2.1, 1.2.2, 1.2.3, 1.2.4, 1.2.5, 1.2.6, 1.2.7, 1.2.8, 1.2.9, 1.2.10, 1.2.11, 1.2.12, 1.2.13, 1.2.14, 1.2.15, 2.0.0-beta1, 2.0.0-beta2, 2.0.0, 2.0.1	62
Chukwa	0.4.0, 0.5.0, 0.5.0-rc0, 0.5.0-rc1, 0.5.0-rc2, 0.5.0-rc3	33
Hadoop	0.1.0, 0.1.1, 0.2.0, 0.2.1, 0.3.0, 0.3.1, 0.4.0, 0.5.0, 0.6.0, 0.6.1, 0.6.2, 0.7.0, 0.7.1, 0.8.0, 0.9.0, 0.9.1, 0.9.2, 0.10.0, 0.10.1, 0.11.0, 0.11.1, 0.11.2, 0.12.0, 0.12.1, 0.12.3, 0.13.0, 0.13.1, 0.14.0, 0.14.1, 0.14.2, 0.14.3, 0.14.4, 0.15.0, 0.15.1, 0.15.2, 0.15.3, 0.16.0, 0.16.1, 0.16.2, 0.16.3, 0.16.4, 0.17.0, 0.17.1, 0.17.2, 0.18.0, 0.18.1, 0.18.2, 0.18.3, 0.19.0, 0.19.1, 0.19.2, 0.20.0, 0.20.1, 0.20.2, 1.0.0, 1.0.1, 1.0.2, 1.0.3, 1.0.4, 1.1.0, 1.1.1, 1.1.2, 1.2.0, 1.2.1	62
Ivy	2.0.0, 2.1.0, 2.2.0, 2.3.0	31
JackRabbit	1.0.1, 1.1.0, 1.1.1, 1.2.1, 1.2.2, 1.2.3, 1.3.0, 1.3.1, 1.3.3, 1.3.4, 1.4.0, 1.5.0, 1.5.2, 1.5.3, 1.5.4, 1.5.5, 1.5.6, 1.5.7, 1.6.0, 1.6.1, 1.6.2, 1.6.4, 1.6.5, 2.0.0-beta1, 2.0.0-beta3, 2.0.0-beta4, 2.0.0-beta5, 2.0.0-beta6, 2.0.0, 2.0.3, 2.0.5, 2.1.0, 2.1.1, 2.1.2, 2.1.3, 2.1.5, 2.1.6, 2.2.0, 2.2.1, 2.2.2, 2.2.4, 2.2.5, 2.2.7, 2.2.8, 2.2.9 2.2.10, 2.2.11, 2.2.12, 2.2.13, 2.3.0, 2.3.2, 2.3.3, 2.3.4, 2.3.5, 2.3.6, 2.3.7, 2.4.0, 2.4.1, 2.4.2, 2.4.3, 2.4.4, 2.4.5, 2.5.0, 2.5.1, 2.5.2, 2.5.3, 2.6.0, 2.6.1, 2.6.2, 2.6.3, 2.6.4, 2.6.5, 2.7.0, 2.7.1, 2.7.2, 2.7.3, 2.7.4	56
Jena	2.7.1, 2.7.2, 2.7.3, 2.7.4, 2.10.0, 2.10.1, 2.11.0, 2.11.1, 2.11.2, 2.12.0	100
JSPWiki	1.4.0, 1.5.0, 1.7.0, 1.8.0, 1.8.2, 2.2.28, 2.2.33, 2.4.69, 2.4.71, 2.4.100, 2.4.102, 2.4.103, 2.4.104, 2.6.0-rc1, 2.6.0, 2.6.1, 2.6.2, 2.6.3, 2.8.0, 2.8.1, 2.8.2, 2.8.3, 2.8.0-alpha1, 2.8.0-beta1	93
Log4j	1.0.4, 1.1.3, 1.2.1, 1.2.2, 1.2.3, 1.2.4, 1.2.5, 1.2.6, 1.2.7, 1.2.8, 1.2.9, 1.2.11, 1.2.12, 1.2.13, 1.2.14, 1.2.15, 1.2.16, 1.2.17, 1.3.0-alpha1, 1.3.0-alpha3, 1.3.0-alpha5, 1.3.0-alpha6, 1.3.0-alpha7, 2.0.0-alpha1, 2.0.0-alpha2, 2.0.0-beta1, 2.0.0-beta2, 2.0.0-beta3, 2.0.0-beta4, 2.0.0-beta5, 2.0.0-beta6, 2.0.0-beta7, 2.0.0-beta8, 2.0.0-beta9, 2.0.0-rc1, 2.0.0-rc2, 2.0.0, 2.0.1, 2.0.2	80
Lucene	3.2.0, 3.3.0, 3.4.0, 3.5.0, 3.6.0, 4.0.0-alpha, 4.0.0-beta, 4.0.0, 4.3.0, 4.3.1, 4.4.0, 4.5.0, 4.5.1, 4.6.0, 4.6.1	22
Mina	0.8.0, 0.8.1, 0.8.2, 0.8.3, 0.8.4, 1.0.0, 1.0.1, 1.0.2, 1.0.3, 1.0.4, 1.0.5, 1.0.8, 1.0.9, 1.0.10, 1.1.0, 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, 1.1.6, 1.1.7, 2.0.0-M1, 2.0.0-M2, 2.0.0-M3, 2.0.0-M4, 2.0.0-M5, 2.0.0-M6, 2.0.0-rc1, 2.0.0, 2.0.1, 2.0.2, 2.0.3, 2.0.4, 2.0.5, 2.0.6, 2.0.7, 2.0.8	93
PDFBox	1.0.0, 1.1.0, 1.2.0, 1.2.1, 1.3.1, 1.4.0, 1.5.0, 1.6.0, 1.7.0, 1.7.1, 1.8.0, 1.8.1, 1.8.2, 1.8.3, 1.8.4, 1.8.5, 1.8.6	50
Struts2	2.0.5, 2.0.6, 2.0.8, 2.0.9, 2.0.11, 2.0.12, 2.2.1, 2.2.3, 2.3.1, 2.3.3, 2.3.4, 2.3.7, 2.3.8	37
Xerces	1.2.0, 1.2.1, 1.2.2, 1.3.0, 1.4.0, 1.4.1, 1.4.3, 2.0.0, 2.1.0, 2.2.0, 2.2.1, 2.3.0, 2.4.0, 2.5.0, 2.6.0, 2.7.0, 2.8.0, 2.9.0	45

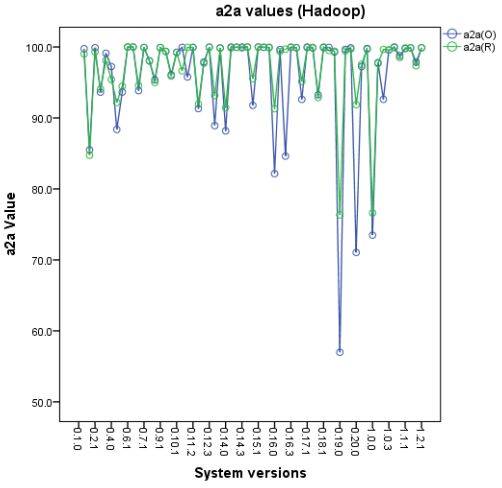
Appendix D

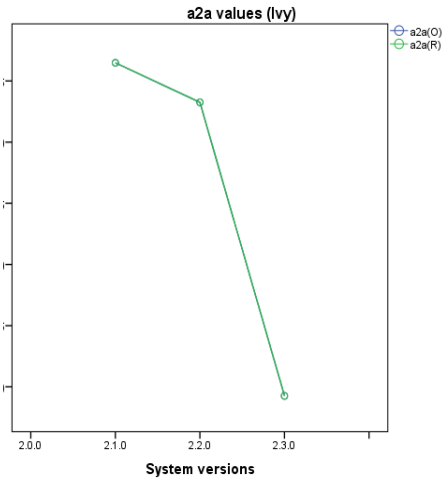

a2a values(Cassandra)
a2a(O)
a2a(R)
a2a Value
System versions

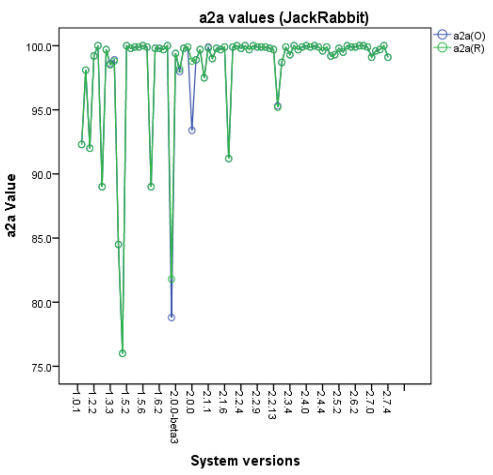

a2a values (Chukwa)
a2a(O)
a2a(R)
a2a Value
0.4.0
0.5.0-rc0
0.5.0-rc1
0.5.0-rc2
0.5.0-rc3
0.5.0
System versions

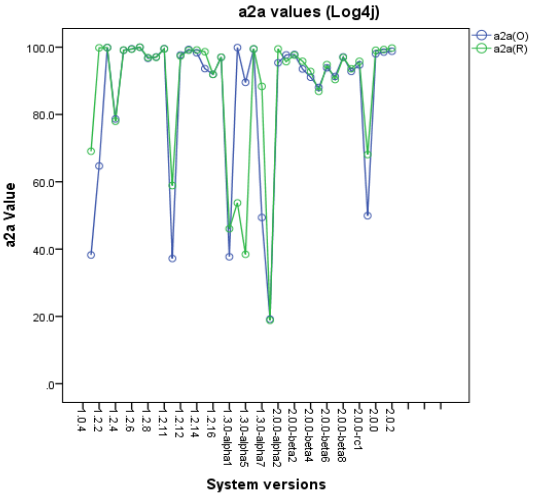

a2a values (Jena)
a2a(O)
a2a(R)
a2a Value
2.7.1
2.7.2
2.7.3
2.7.4
2.10.0
2.10.1
2.11.0
2.11.1
2.11.2
2.12.0
System versions

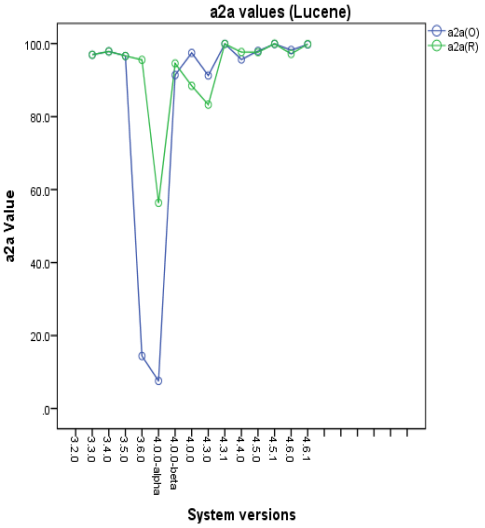

a2a values(JSPWiki)
a2a(O)
a2a(R)
System versions

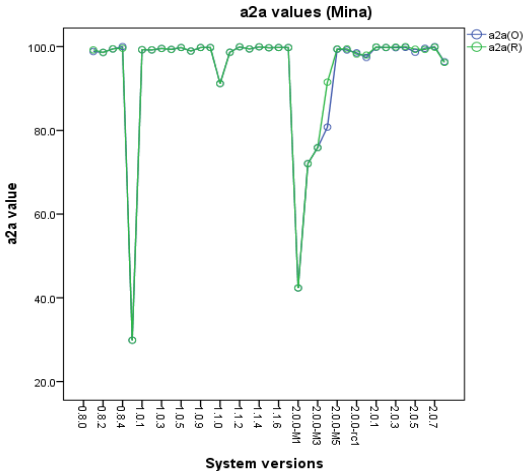

a2a values (Hadoop)
a2a(O)
a2a(R)
a2a Value
System versions

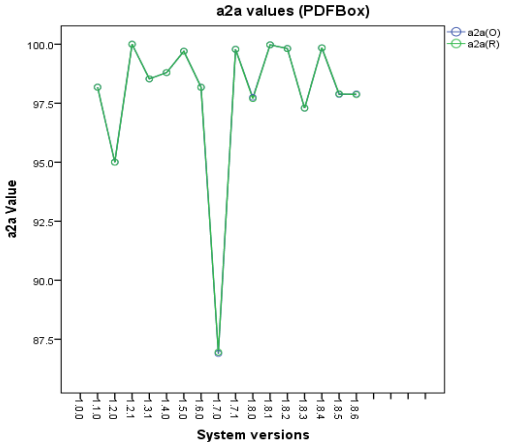

a2a values (Ivy)
a2a(O)
a2a(R)
2.0.0
2.1.0
2.2.0
2.3.0
System versions

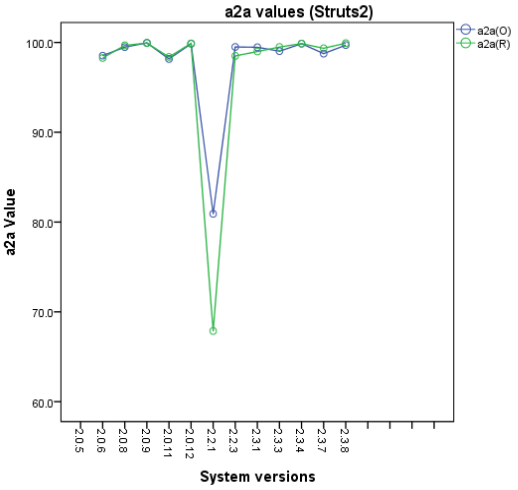

a2a values (JackRabbit)
a2a(O)
a2a(R)
a2a Value
System versions

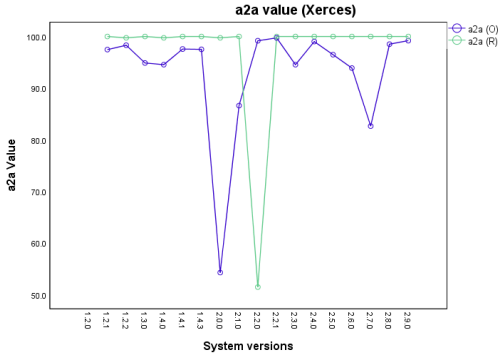

a2a values (Log4j)
a2a(O)
a2a(R)
a2a Value
System versions


a2a values (Lucene)
a2a(O)
a2a(R)
a2a Value
System versions

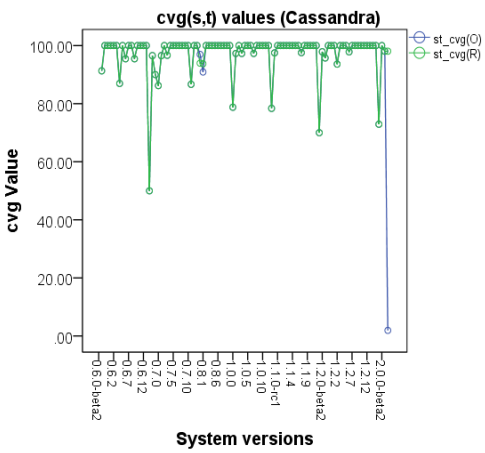

a2a values (Mina)
a2a(O)
a2a(R)
a2a value
System versions

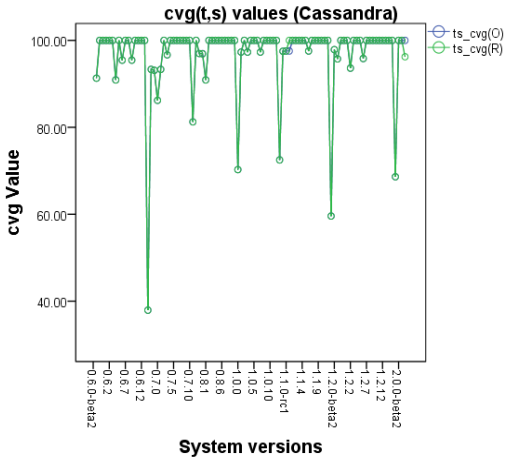

a2a values (PDFBox)
a2a(O)
a2a(R)
a2a Value
System versions

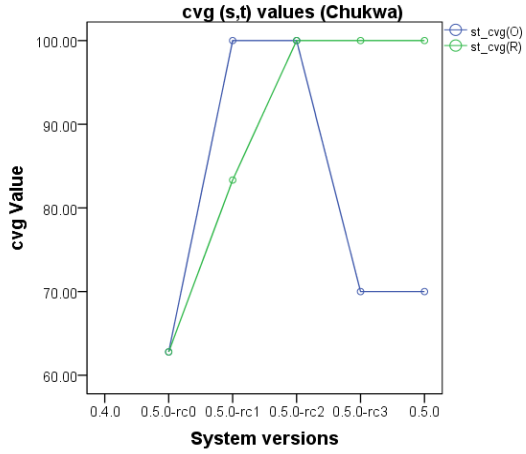

a2a values (Struts2)
a2a(O)
a2a(R)
a2a Value
System versions

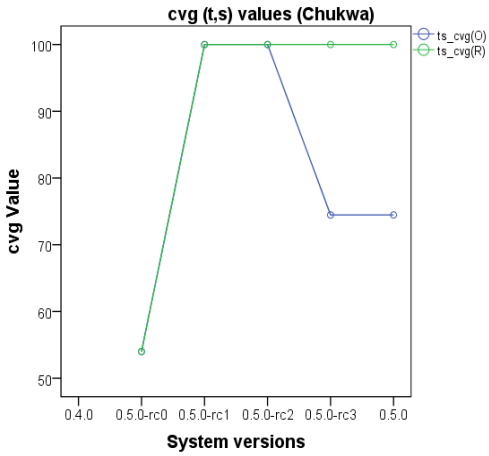

a2a value (Xerces)
a2a (O)
a2a (R)
a2a Value
System versions

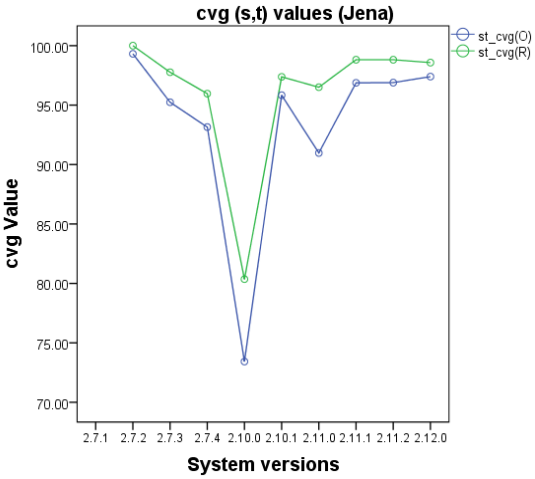
Appendix E

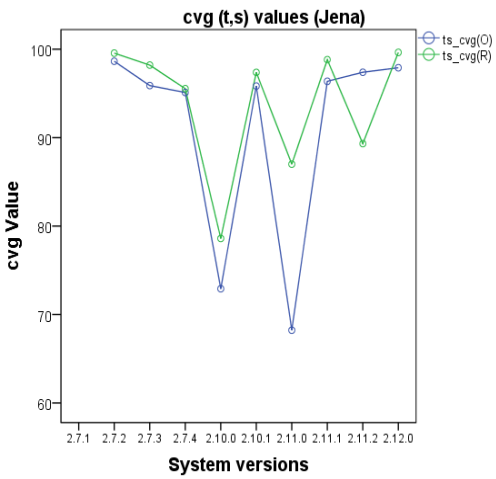

cvg(s,t) values (Cassandra)
st_cvg(O)
st_cvg(R)
cvg Value
System versions

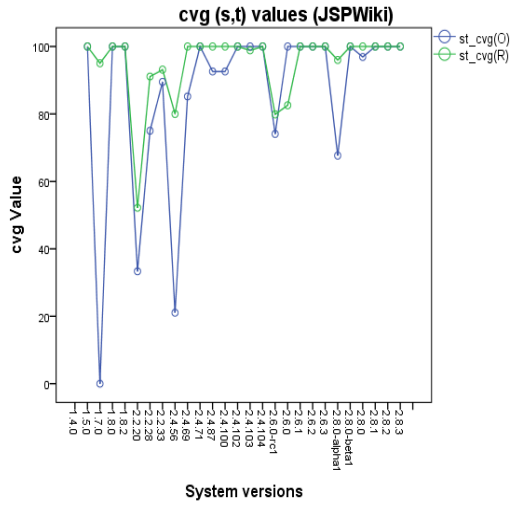

cvg(t,s) values (Cassandra)
ts_cvg(O)
ts_cvg(R)
cvg Value
System versions

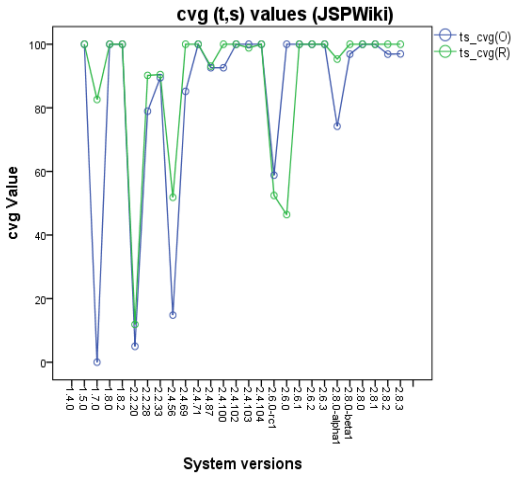

cvg (s,t) values (Chukwa)
st_cvg(O)
st_cvg(R)
cvg Value
System versions

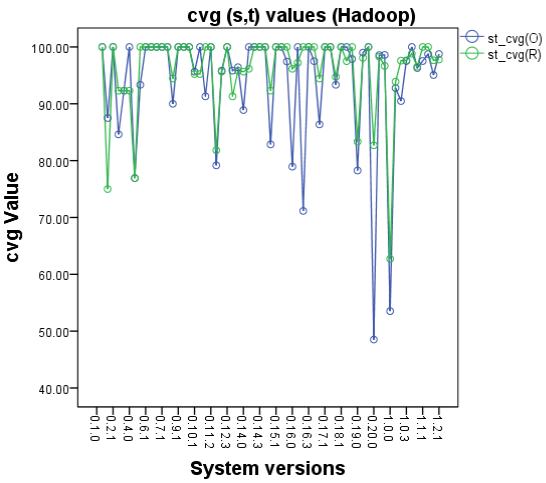

cvg (t,s) values (Chukwa)
ts_cvg(O)
ts_cvg(R)
cvg Value
System versions

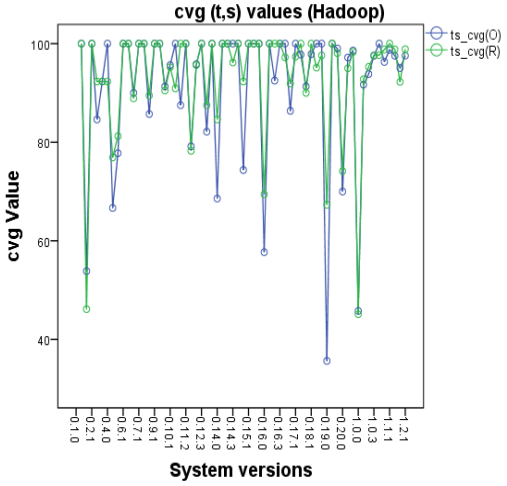

cvg (s,t) values (Jena)
st_cvg(O)
st_cvg(R)
cvg Value
System versions

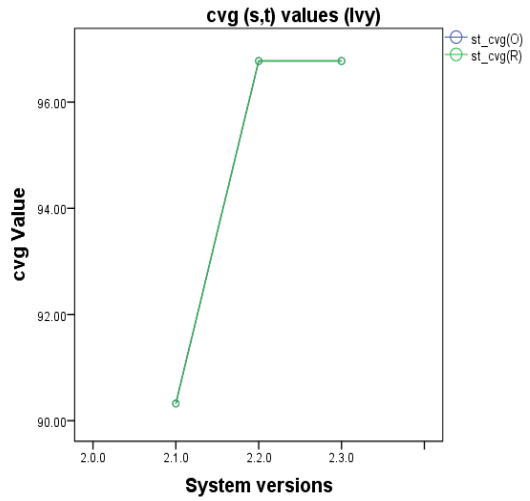

cvg (t,s) values (Jena)
ts_cvg(O)
ts_cvg(R)
cvg Value
System versions

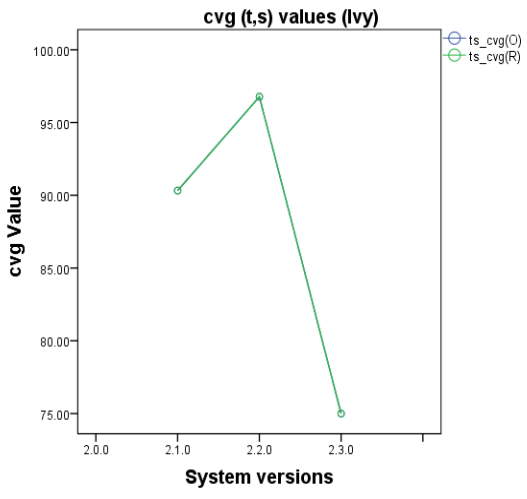

cvg (s,t) values (JSPWiki)
st_cvg(O)
st_cvg(R)
cvg Value
System versions

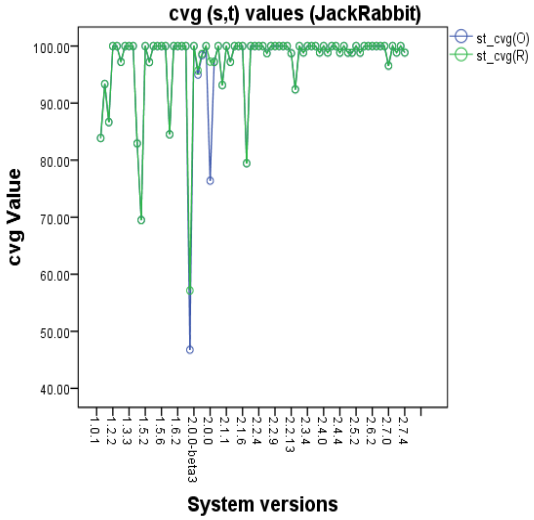

cvg (t,s) values (JSPWiki)
ts_cvg(O)
ts_cvg(R)
cvg Value
System versions

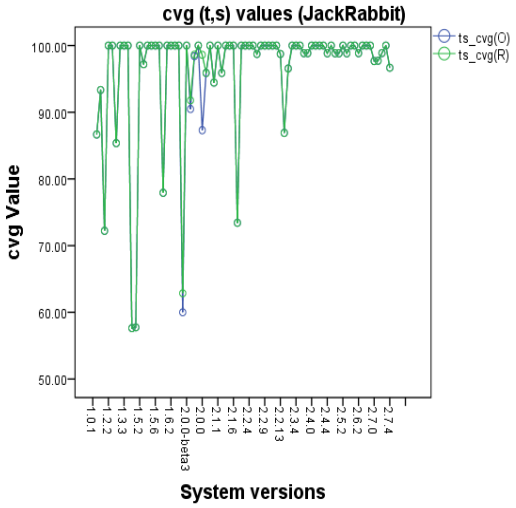

cvg (s,t) values (Hadoop)
st_cvg(O)
st_cvg(R)
cvg Value
System versions

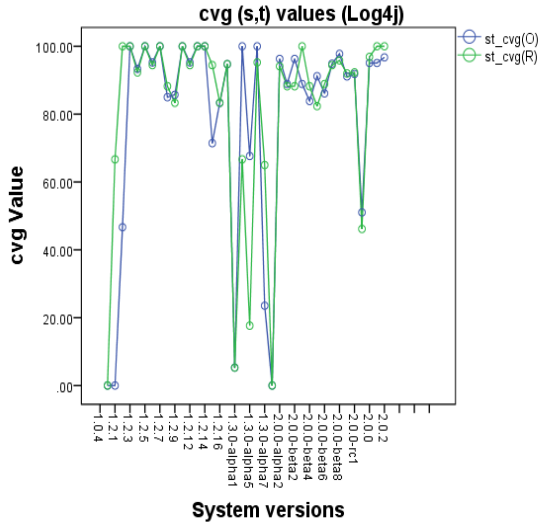

cvg (t,s) values (Hadoop)
ts_cvg(O)
ts_cvg(R)
cvg Value
System versions

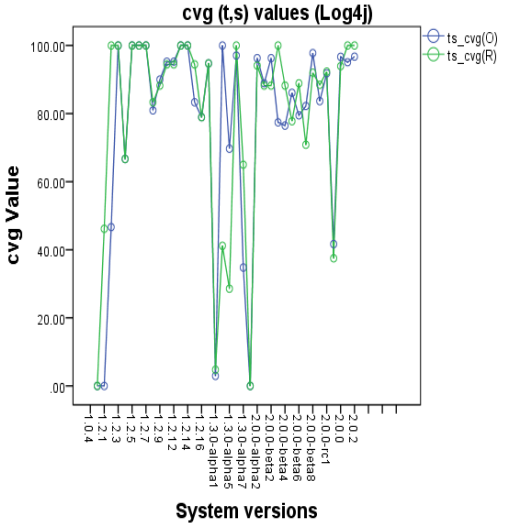

cvg (s,t) values (Ivy)
st_cvg(O)
st_cvg(R)
cvg Value
System versions

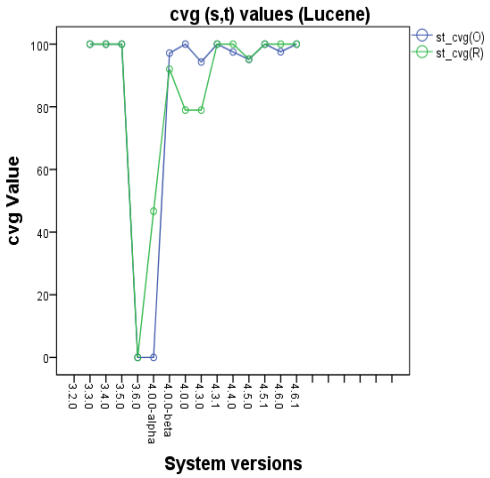

cvg (t,s) values (Ivy)
ts_cvg(O)
ts_cvg(R)
cvg Value
System versions

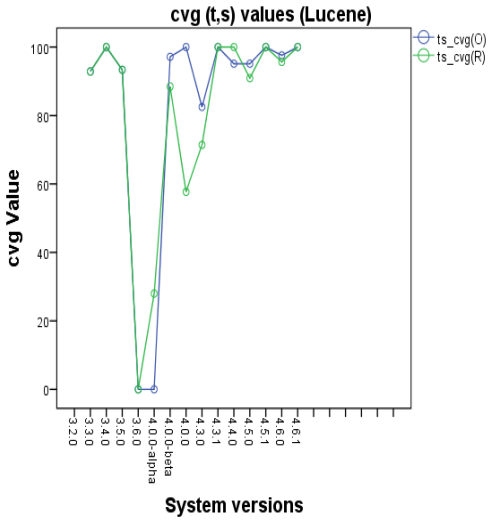

cvg (s,t) values (JackRabbit)
st_cvg(O)
st_cvg(R)
cvg Value
System versions

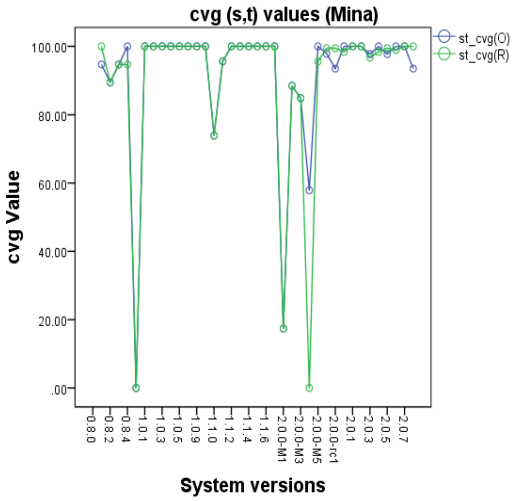

cvg (t,s) values (JackRabbit)
ts_cvg(O)
ts_cvg(R)
cvg Value
System versions

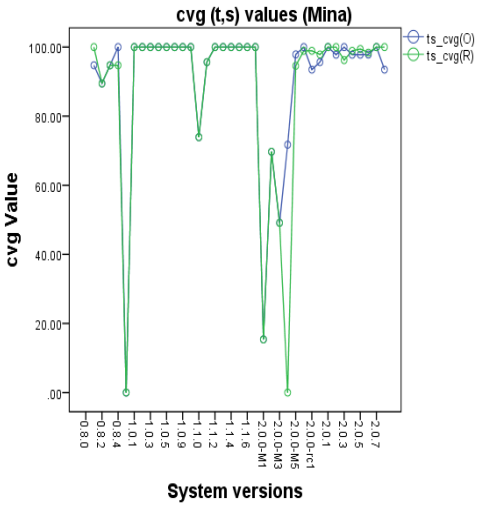

cvg (s,t) values (Log4j)
st_cvg(O)
st_cvg(R)
cvg Value
System versions

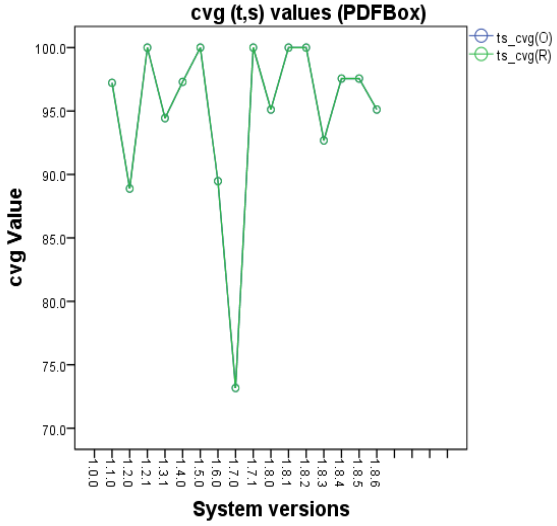

cvg (t,s) values (Log4j)
ts_cvg(O)
ts_cvg(R)
cvg Value
System versions

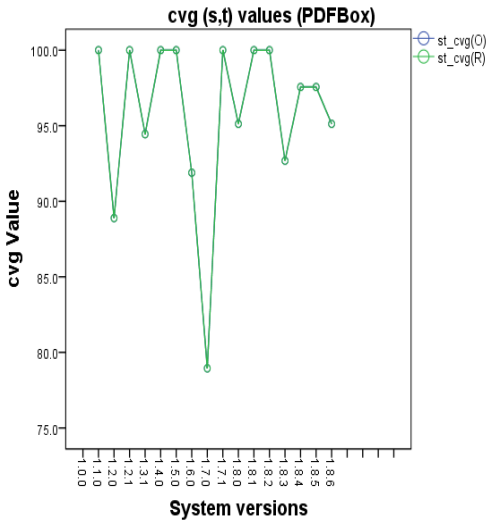

cvg (s,t) values (Lucene)
st_cvg(O)
st_cvg(R)
cvg Value
System versions

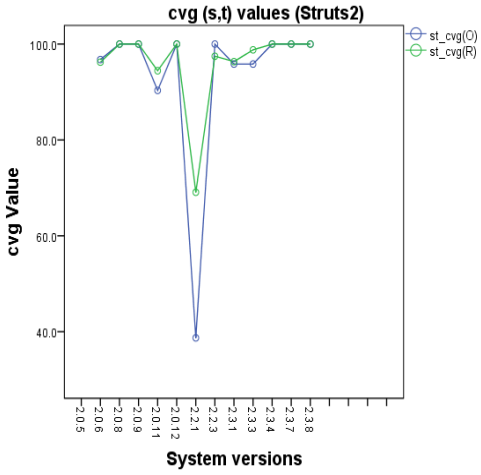

cvg (t,s) values (Lucene)
ts_cvg(O)
ts_cvg(R)
cvg Value
System versions

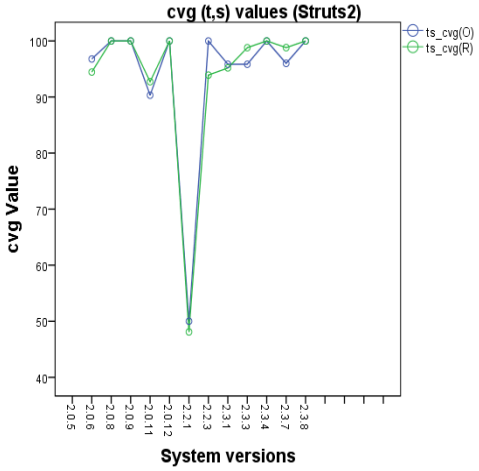

cvg (s,t) values (Mina)
st_cvg(O)
st_cvg(R)
cvg Value
System versions

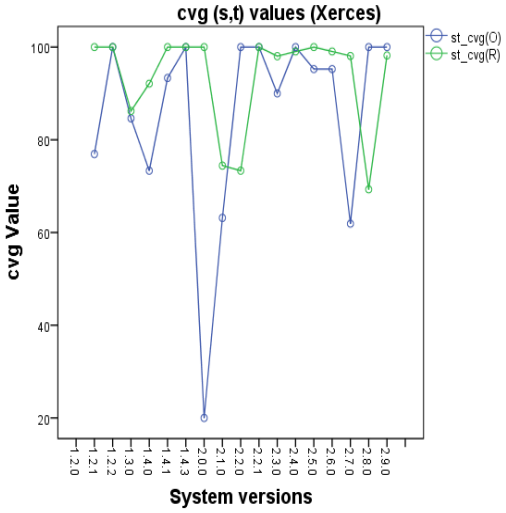

cvg (t,s) values (Mina)
ts_cvg(O)
ts_cvg(R)
cvg Value
System versions

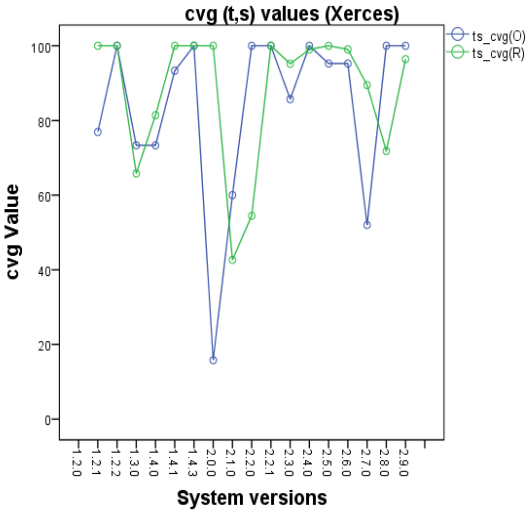

cvg (t,s) values (PDFBox)
ts_cvg(O)
ts_cvg(R)
cvg Value
System versions


cvg (s,t) values (PDFBox)
st_cvg(O)
st_cvg(R)
cvg Value
System versions


cvg (s,t) values (Struts2)
st_cvg(O)
st_cvg(R)
cvg Value
System versions


cvg (t,s) values (Struts2)
ts_cvg(O)
ts_cvg(R)
cvg Value
System versions


cvg (s,t) values (Xerces)
st_cvg(O)
st_cvg(R)
cvg Value
System versions


cvg (t,s) values (Xerces)
ts_cvg(O)
ts_cvg(R)
cvg Value
System versions